Research

Identification of Behavioral and Creational Design Motifs through Dynamic Analysis

Janice Ka-Yee Ng, Yann-Gaël Guéhéneuc*, and Giuliano Antoniol

Ptidej Team and SOCCER lab, Département de génie informatique et génie logiciel École Polytechnique de Montréal, CP 6079 succ. Centre Ville, Montréal, Québec, H3C 3A7, Canada

SUMMARY

Design patterns offer design motifs, solutions to object-oriented design problems. Design motifs lead to well-structured designs and thus are believed to ease software maintenance. However, after use, they are often "lost" and are consequently of little help during program comprehension and other maintenance activities. Therefore, several works proposed design pattern identification approaches to recover occurrences of the motifs. These approaches mainly used the structure and organisation of classes as input. Consequently, they have a low precision when considering behavioural and creational motifs, which pertain to the assignment of responsibilities and the collaborations among objects at runtime. We propose MoDeC, an approach to describe behavioral and creational motifs as collaborations among objects in the form of scenario diagrams. We identify these motifs using dynamic analysis and constraint programming. Using a proof-of-concept implementation of MoDeC and different scenarios for five other Java programs and Builder, Command, and Visitor, we show that MoDeC has a better precision than a state-of-the-art static approaches.

KEY WORDS: Reverse engineering; Design pattern identification; Scenario diagrams; Constraint programming; Dynamic analysis.

*Correspondence to: Yann-Gaël Guéhéneuc, yann-gael.gueheneuc@polymtl.ca

1. Introduction

Software maintenance is a crucial phase of the object-oriented software development process. Indeed, it can take as much as 90% of the total resources dedicated to a program [9, 33]. Program comprehension is the main activity of maintenance. During program comprehension, developers must understand the structure and organisation of classes. They typically use code reading techniques or reverse-engineering tools to analyse and relate classes before performing other maintenance activities, such as debugging or adding new features. Program comprehension is facilitated significantly if the program possesses a well-structured design.

Since their inception in 1995, design motifs—the solutions offered by design patterns [12]—have been increasingly used in object-oriented development to obtain well-structured designs. They pertain to the *structure* and organisation of classes and to the *creation* and *behaviour* of objects at runtime. However, after use, they are often "lost" in the source code because their responsibilities are scattered over several classes and their implementation are often not explicitly documented. Therefore, they cannot be readily used during program comprehension.

Several approaches have been proposed to identify occurrences of micro-architectures similar to structural design motifs using static analysis [16, 24, 26, 27, 30, 34, 40, 43]. For example, we proposed DeMIMA [16], a Design Motif Identification Multi-layered Approach, which uses explanation-based constraint programming to identify structural motifs in program models reverse-engineered from source code. DeMIMA has a good accuracy; yet, as other approaches, it suffers from a low precision when motifs cannot be exclusively described by their structure.

Behavioral and creational motifs as well as some structural motifs can hardly be described only by the structure and organisation of their classes. They must be described as runtime collaborations among objects, as shown in [21, 22, 42]. Thus, the responsibility of runtime objects should not be neglected because collaborations among these objects, along with the structure of their classes, could increase the precision of the identification process.

Consequently, we propose MoDeC, a semi-automated 3-step approach to identify behavioral and creational design motifs in source code using dynamic analysis. The novelty of MoDeC is three-fold: it uses (1) scenario diagrams of both the programs and the motifs as input; (2) the static occurrences produced by a previous design pattern identification approach to reduce the search space and guide the building of interesting scenarios; and, (3) explanation-based constraint programming to identify occurrences of the motifs in the programs while providing automation and explanations.

In this paper, we use the Memento pattern to illustrate the modelling of motifs, the instrumentation of code, and the generation of a scenario diagram. We also show the practicality of our approach by reusing existing components to implement a proof-of-concept and by performing an empirical validation using this implementation. The empirical validation divides in a case study with the Command motif and JHotDraw, an empirical study using five real-world Java programs and the Builder, Command, and Visitor motifs, and a comparative study with previous work. It shows that MoDeC has a good precision and recall and allows increasing the precision of static occurrences.

This work extends our previous work [37] with a description of our approach using a toy example, a complete case study on JHotDraw; an empirical study; and a comparative study. This paper is organised as follows. Section 2 provides an overview of our approach. Section

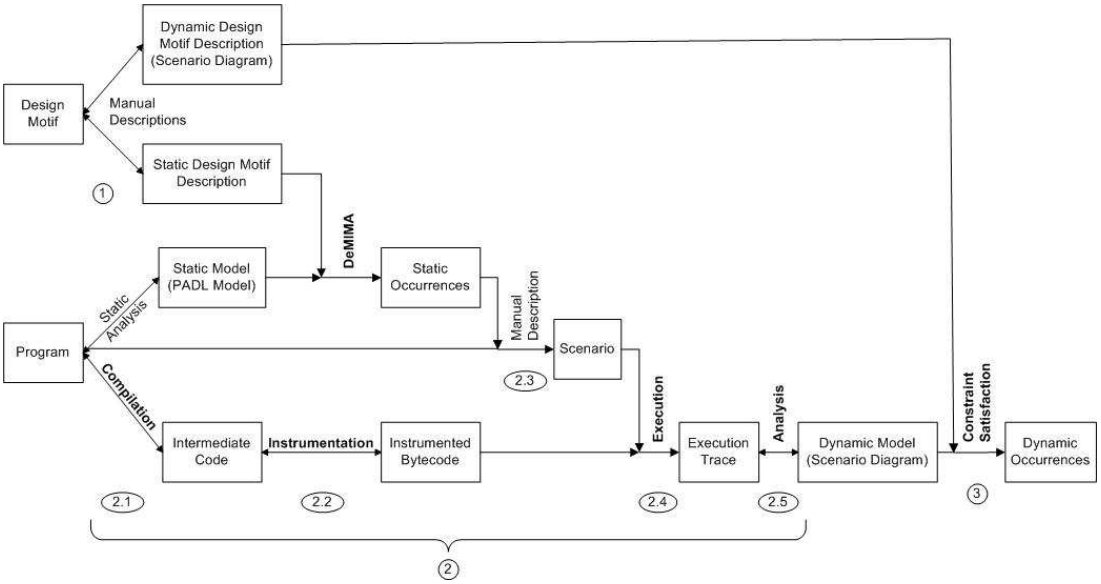
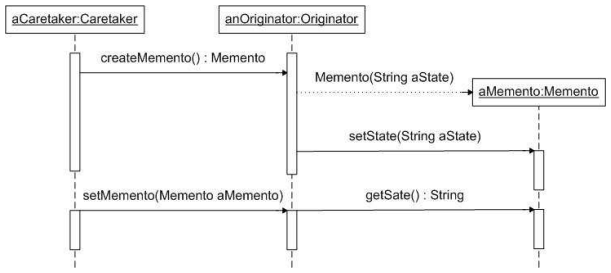


Figure 1. A 3-step approach for design pattern identification through dynamic analysis. Steps in bold are automated.

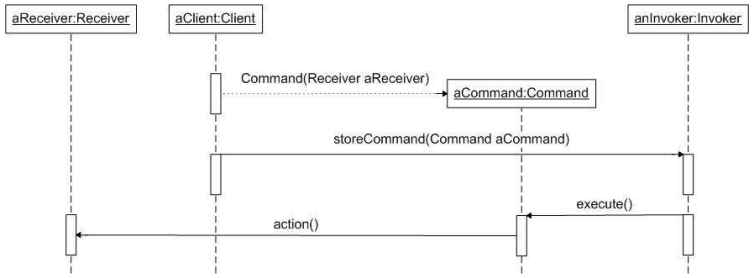
3 introduces a metamodel to capture the collaborations among objects at runtime, which is similar to the UML metamodel for sequence diagrams. It also illustrates the use of the metamodel. Section 4 details our technique to reverse engineer scenario diagrams of object-oriented programs using dynamic analysis. It includes a description of the use of the results of a static approach to build appropriate scenarios. Section 5 elaborates on the technique used to identify behavioral and creational design motifs. Section 6 illustrates MoDeC on JHotDraw and, using five Java programs, reports the results of the identification of the Builder, Command, and Visitor motifs. It also compares MoDeC to previous approaches. Section 8 presents related work. Finally, Section 9 concludes with future work.

2. MoDeC in a Nutshell and Design Rationale

Figure 1 describes the different steps of the MoDeC approach; steps in bold are automated while the others are intrinsically manual. First, a behavioral or creational design motif (or the behavioural/creational part of a mostly-structural motif) is described as a scenario diagram (a subset of a UML-like sequence diagram as explained in Section 3). Second, using dynamic analysis, a dynamic model of the collaborations among the objects of a program is reverse-engineered for a given scenario, also in the form of a scenario diagram. Motifs are modelled as collaborations among objects and their operations, *i.e.*, scenario diagrams, as are the behavior of programs, to unify the representation of both motifs and programs and to subsequently ease



(a) The Memento motif



(b) The Command motif

Figure 2. Description of motifs in terms of collaborations.

the dynamic identification process. The scenario used to reverse engineer the dynamic model is built from the *static*† occurrences identified by a static approach, DeMIMA. Finally, the identification process is translated into a constraint satisfaction problem (CSP). By solving the CSP, concrete objects and operations from the program scenario diagram are assigned to the roles in the motif scenario diagram. We thus obtain the *dynamic* occurrences of the motif. MoDeC is semi-automated. The user's input is required (1) to describe motifs, (2) to design, and (3) to exercise appropriate scenarios to generate traces. The first user's input, the motif description, is the result of a short activity performed only once because each motif is kept in a repository for later use. Automation is provided to identify static occurrences using DeMIMA and to apply the constraint solver to identify dynamic occurrences.

In the rest of this paper, we illustrate the details of each step of MoDeC using, as running example, a toy program, which implements the Memento motif. The Memento design pattern

†We use the adjectives *static* and *dynamic* to distinguish occurrences identified by DeMIMA and MoDeC, respectively.

provides a solution to the problem of capturing and exporting the internal state of some object, so that the object can be restored to this state later, without violating encapsulation. Figure 2(a) is a scenario diagram illustrating the collaborations among the classes and objects participating in the Memento motif. A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator. The caretaker may never pass the memento back to it originator, if it never needs to revert to an earlier state. Mementos are passive, only the originator that created a memento can assign or retrieve its state.

We could use a static or a dynamic analysis to capture the behavior of programs and build their scenario diagrams. Each type of analyses has advantages and drawbacks. On the one hand, a static analysis would provide a complete sequence diagram (set of scenario diagrams) of a program. However, it would not reflect its *real* behaviour and would require to analyse source code to determine the dynamic types of object references, which is not conceivable for large programs [13]. Moreover, static approaches can have 100% recall at the price of a low precision. On the other hand, a dynamic analysis provides only *parts* of the whole behaviour of a program but reports precisely the collaborations among objects. We seek to improve precision and thus choose to use dynamic analysis.

The programming language in which the programs are written does not impact the principles of our approach but impacts the instrumentation strategy [4]. There are several possible strategies to retrieve data by dynamic analysis. These strategies include source-code, intermediate-code, and virtual-machine instrumentation or the use of a customised debugger. We choose to instrument the intermediate code to avoid maintaining a "clean" *and* an instrumented version of the same source code and handling inconsistencies between the two versions. Also, intermediate-code instrumentation is less intrusive than instrumenting a virtual machine or customising a debugger. The limitations of this strategy are its specificity to the target language and the additional delays in the program executions. We accept these two limitations because they do not undermine our approach. Following [4], we instrument intermediate code to trace method calls at runtime and translate these calls into sequences of operations. We do not trace "patterns" of execution and conditions under which operations are executed because they are unnecessary in our approach.

The implementation of our instrumentation strategy consists of five sub-steps, illustrated in Figure 1, Steps 2.1 to 2.5. First, we compile the source files of a program to obtain its intermediate code. Second, we instrument the intermediate code. Third, we choose a scenario to be executed. The choice of the scenario is guided by the documentation of the program, where a list of functionalities is described, possibly including terms found in the descriptions of some motifs (their intents and motivations principally) and by the static occurrences identified by a static identification approach, such as DeMIMA. Fourth, we execute the instrumented program following the chosen scenario to produce an execution trace. Finally, we analyse the trace and instantiate a scenario diagram. We illustrate this strategy on Java in Section 4 after presenting the meta-model of a scenario diagram in the following section.

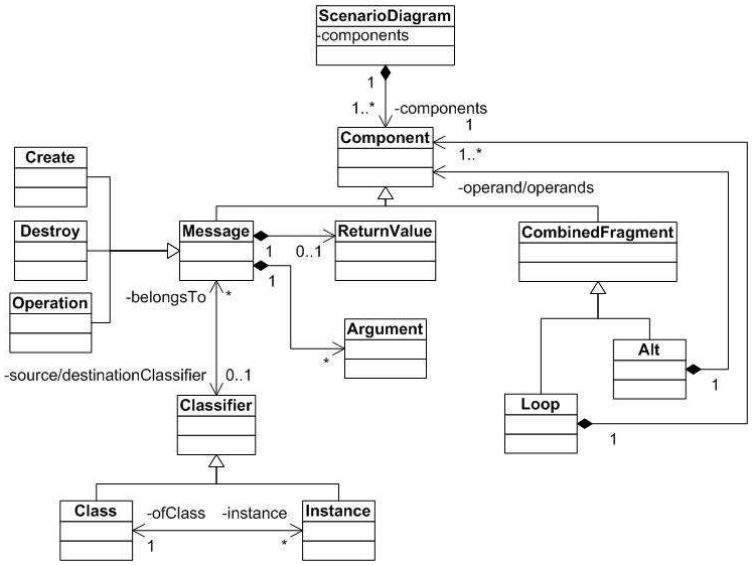


Figure 3. Scenario diagram metamodel following UML naming convention.

3. Scenario Diagrams of Behavioural and Creational Design Motifs

We model behavioral and creational design motifs as collaborations among objects. We use scenario diagrams to express these collaborations because they describe the objects and their operations according to the scenario for which the motifs bring design solutions. We first describe the scenario diagram metamodel [39] and then discuss its use and illustrate its instantiation on the running example.

3.1. Metamodel of Scenario Diagrams

In [12], some design motifs are provided with representations in the form of collaborations among objects. Gamma *et al.* chose to use diagrams similar to scenario diagrams to show the sequences of operations among objects, *i.e.*, the order in which operations among the objects participating in the motifs are executed.

Therefore, our metamodel (cf. Figure 3) focuses on describing control flow, *i.e.*, the sequence of *operations* passed among *objects* at runtime, and allows describing a *scenario diagram* of a motif, *i.e.*, a UML sequence diagram obtained from the execution of one particular use case [4]: the *one* scenario for which the motif brings a solution. A scenario diagram is only a partial UML sequence diagram, describing one scenario instead of all possible alternatives for the exercised use case.

A scenario diagram, class `ScenarioDiagram` on Figure 3, contains an ordered list of components, class `Component`, that can either be messages, class `Message`, or combined fragments, class `CombinedFragment`. Messages can be of three different types: an operation,

class `Operation`, a destruction, class `Destroy`, or a creation, class `Creation`. Messages have a `sourceClassifier` and a `destinationClassifier` to represent the concept of caller and callee. Caller and callee are of type `Classifier` and are specialised as either an `Instance` or a `Class`, the latter case being applicable if the message represents a class-level operation. If appropriate, messages include arguments, class `Argument`, of different types: either primitive types or object types. The return value of messages is represented by class `ReturnValue`. (In the following, we use "operation" as a synonym for "message".)

The class `CombinedFragment` is inspired by the UML notation [28] to group sets of operations to show conditional flows in sequence diagrams. Although [28] provides 11 interaction types of combined fragments, we performed a study of the motifs defined in [12] that showed that only the combined fragments *loop* and *alternative* are necessary to the identification of behavioral and creational motifs. Consequently, combined fragments are specialised into two types: loops, class `Loop`, to illustrate repetitions of operations and alternatives, class `Alt`, to describe mutually exclusive choices among sequences of operations. The composition relationships `operand` between classes `Loop` and `Component` and between classes `Alt` and `Component` allow to model nested loops and–or alternatives. A loop has one and only one operand, while an alternative has one or more operands. For example, the classic alternative *if then else* has two operands: operand *if* and operand *else*.

This metamodel defines a class of languages: each modelled design motif defines a language that is used to generate an *acceptor* of this language. The acceptor takes as input a trace, also in the form of a scenario diagram, and outputs sequences of the input matching the language, *i.e.*, occurrences of the motif. Within this perspective, our metamodel can be classified as a Type-3 grammar (regular grammar) that generates regular languages. The metamodel is extensible and it is thus possible to add new interactions types in the future.

3.2. Illustration of a Scenario Diagrams

In our approach, see Figure 1, Step 1, behavioral and creational design motifs are described by translating manually the collaborations suggested in a motif into an instance of the scenario diagram metamodel. For example, for each operation involved in the diagram of the Memento motif, in Figure 2(a), we instantiate an object `Operation` that is added to the ordered list `components` of an instance of `ScenarioDiagram` representing the Memento motif. Given operation `setMemento(Memento aMemento)` takes an object of type `Memento` as argument, we instantiate an object `Argument`, whose attribute `type` points to `Memento`. This object `Argument` is added to the ordered list `arguments` of operation `setMemento(Memento aMemento)`. The participants collaborating in the motif, `aCaretaker`, `anOriginator`, and `aMemento`, are instantiated as instances of `Instance`, and are set to be the `sourceClassifier` or `destinationClassifier` of the corresponding operations. The `sourceClassifier` of `createMemento()` and `setMemento(Memento aMemento)` is `aCaretaker`, while `anOriginator` is their `destinationClassifier`.

Similarly to structural design motifs, behavioural and creational motifs must be adapted by the developers to the context in which they are used. Consequently, the suggested collaborations may not be implemented strictly. Our validation led us to identify two recurring variants in the concrete collaborations with respect to the suggested collaborations. First, we

⟨*trace*⟩ → ⟨*event*⟩ | ⟨*event*⟩ ⟨*trace*⟩ | ϵ
⟨*event*⟩ → ⟨*message_type*⟩ ⟨*message_state*⟩ ⟨*message_signature*⟩ **callee** ⟨*class_identifier*⟩ {⟨[*id*]⟩}
⟨*message_type*⟩ → **operation** | **constructor** | **destructor**
⟨*message_state*⟩ → **start** | **end**
⟨*message_signature*⟩ → {⟨*visibility*⟩} {**static**} ⟨*return_type*⟩ ⟨*message_identifier*⟩ **(** {⟨*argument*⟩} **)**
⟨*visibility*⟩ → **public** | **private** | **protected**
⟨*message_identifier*⟩ → *name of the message being called*
⟨*argument*⟩ → ⟨*argument_type*⟩ ⟨*argument_identifier*⟩ {**,** ⟨*argument*⟩} | ϵ
⟨*argument_type*⟩ → *type of the argument*
⟨*argument_identifier*⟩ → *name of the argument*
⟨*class_identifier*⟩ → *name of the class to which belongs the message being called*
⟨*id*⟩ → *unique number of the instance object to which belongs the message being called*

Figure 4. The trace format.

observed that it is a common practice to add intermediate objects and operations to the original collaborations while implementing a motif. Second, we also observed that collaborations are often implemented by more than one operation.

For example, in the description of the Command motif shown in Figure 2(b), the `Command` object invokes the unique operation `action()` on its receiver to carry out a request. This operation will often correspond to several concrete operations in real programs. Indeed, it is unlikely that *one* operation will be sufficient to carry out the request of the invoker. Instead, a set of operations will be called to complete the request.

Therefore, in the descriptions of the motifs, we immediately consider these two variants because they neither change the structure characterising the motifs nor the semantics of their original collaborations but reflect concrete implementations. However, we do not vary more our descriptions. Instead, constraints on the collaborations among objects will be relaxed based on the descriptions, as further explained in Section 5.

4. Scenario Diagrams of Programs

We also model the dynamic behaviours of programs as scenarios diagrams. Following our approach shown in Figure 1, we now describe the model of execution traces used in Step 2.4, the instrumentation of programs to generate such traces in Step 2.2, the building of appropriate scenarios to exercise the programs in Step 2.3, and the combination of these steps to obtain scenario diagrams in Step 2.5.

4.1. Execution Trace

In MoDeC, each execution trace must comply with the syntax shown in Figure 4. A trace contains a set of events. Each event corresponds to either the `start` or the `end` of an operation,

so that we can infer nested operations. Each event records the name of the operation, its formal arguments, as well as the unique ID of the callee, *i.e.*, receiver object. Traces are independent of the programming language, program, and scenarios and could be generated for any object-oriented programming language.

4.2. Instrumentation

We instrument Java byte-code with BCEL—the Byte Code Engineering Library [3], which offers the possibility to create, analyse, and manipulate Java class files easily. Following the definition of the metamodel and its three types of messages, we instrument constructor and method executions.

We now detail the instrumentation of constructors. Constructors appear in byte-code as instance methods with a special name `<init>`. Figure 5 illustrates the constructor of class `Memento` from the Memento motif shown in Figure 2(a). Its byte-code before instrumentation is shown in Figure 6.

We insert byte-code instructions, lines 3–5 in Figure 7, after the invocation of the superclass constructor so that a `constructor start` event is written in the execution trace. Then, we identify every `return` byte-code instruction and calls to `System.exit(int)` to write a `constructor end` event in the execution trace, lines 19–21 in Figure 7, and thus trace corresponding `constructor end` events.

Each time a `constructor start` event or `constructor end` event is written, we also write in the trace the `destinationClassifier` class name and the object identity hash-code to identify each object in a program uniquely. The corresponding byte-code instructions are shown on lines 10–15 and 26–31 in Figure 7. The `sourceClassifier` is determined when instantiating the scenario diagram, as discussed in Section 4.4. Multi-threading is not an issue in our approach because (1) each thread holds the lock to the trace file upon writing and (2) in our experience, objects playing a role in a motif are always called within the same thread. Consequently, interleaved messages would only be an issue if a strict ordering of messages was required and if two objects playing a role in a motif were accessed by two different threads.

The generation of `destruction` events from the execution of Java programs is difficult because Java does not provide explicit destructors, as C++ does for example. It only allows `finalize()` methods that may or may not be called by the Java virtual machine when the garbage collector collects objects, as we discussed in our previous study [17].

4.3. Building a Scenario

The choice of the scenario to run the program to collect its execution trace is important because it should exercise as many objects as possible from classes most likely to implement the design motif of interest. The problem of building scenarios and choosing the most appropriate scenario to exercise certain classes, objects, methods, or functions of a program has been previously studied in previous works, for example [6, 7, 8, 14, 38]. We get inspiration from these works to avoid building arbitrary scenarios and to reduce the large space of all possible scenarios for any non-trivial program: we use the static occurrences obtained from DeMIMA to identify

```
1  public class Memento {
2
3     public Memento (String state) {
4          this.setState(state);
5     }
6  }
```

Figure 5. Partial source code of class `Memento`.

```
1  aload_0
2  invokespecial Object.<init> ()V
3  aload_0
4  ldc           "state"
5  putfield      Originator.state String;
6  return
```

Figure 6. Constructor byte-code before instrumentation.

```
1  aload_0
2  invokespecial Object.<init> ()V
3  ldc           "Memento_GOF_EVALUATION.trace"
4  ldc           "constructor start <init> CALLEE Originator"
5  invokestatic  LogToFile.write (Object;)V
6  ldc           "Memento_GOF_EVALUATION.trace"
7  new           <Integer>
8  dup
9  aload_0
10 invokestatic  System.identityHashCode (Object;)I
11 invokespecial Integer.<init> (I)V
12 invokestatic  LogToFile.write (Object;)V
13 ldc           "Memento_GOF_EVALUATION.trace"
14 ldc           "\n"
15 invokestatic  LogToFile.write (Object;)V
16 aload_0
17 ldc           "state"
18 putfield      Originator.state Ljava/lang/String;
19 ldc           "Memento_GOF_EVALUATION.trace"
20 ldc           "constructor end <init> CALLEE Originator"
21 invokestatic  LogToFile.write (Object;)V
22 ldc           "Memento_GOF_EVALUATION.trace"
23 new           <Integer>
24 dup
25 aload_0
26 invokestatic  System.identityHashCode(Object;)I
27 invokespecial Integer.<init> (I)V
28 invokestatic  LogToFile.write(Object;)V
29 ldc           "Memento_GOF_EVALUATION.trace"
30 ldc           "\n"
31 invokestatic  LogToFile.write(Object;)V
32 return
```

Figure 7. Constructor byte-code after instrumentation.

```
operation start public static void main(String[] args) callee ModelMementoTest [-1]
constructor start public void <init>() callee Caretaker [14613018]
constructor start public void <init>() callee Originator [12386568]
constructor end  public void <init>() callee Originator [12386568]
constructor end public void <init>() callee Caretaker [14613018]
operation start public void callCreateMemento() callee Caretaker [14613018]
operation start public Memento createMemento() callee Originator [12386568]
constructor start public void <init>() callee Memento [17237886]
constructor end public void <init>() callee Memento [17237886]
operation start public void setState(String state) callee Memento [17237886]
operation end public void setState(String state) callee Memento [17237886]
operation end public Memento createMemento() callee Originator [12386568]
operation end public void callCreateMemento() callee Caretaker [14613018]
operation start public void undoOperation() callee Caretaker [14613018]
operation start public void setMemento(Memento m) callee Originator [12386568]
operation start public String getState() callee Memento [17237886]
operation end public String getState() callee Memento [17237886]
operation end public void setMemento(Memento m) callee Originator [12386568]
operation end public void undoOperation() callee Caretaker [14613018]
operation end void public static void main (String[] args) callee ModelMementoTest [-1]
```

Figure 8. Execution trace of a toy program implementing the Memento motif.

```
<OPERATION> public static void main (String[] args)
        <CALLEE> ModelMementoTest       <CALLER> inexistant
<CREATE> public void <init>()
        <CALLEE> Caretaker [14613018]     <CALLER> ModelMementoTest
<CREATE> public void <init>()
        <CALLEE> Originator [12386568]    <CALLER> Caretaker [14613018]
<OPERATION> public void callCreateMemento()
        <CALLEE> Caretaker [14613018]     <CALLER> ModelMementoTest
<OPERATION> public Memento createMemento()
        <CALLEE> Originator [12386568]    <CALLER> Caretaker [14613018]
<CREATE> public void <init>()
        <CALLEE> Memento [17237886]       <CALLER> Originator [12386568]
<OPERATION> public void setState(String state)
        <CALLEE> Memento [17237886]       <CALLER> Originator [12386568]
<OPERATION> public void undoOperation()
        <CALLEE> Caretaker [14613018]     <CALLER> ModelMementoTest
<OPERATION> public void setMemento(Memento m)
        <CALLEE> Originator [12386568]    <CALLER> Caretaker [14613018]
<OPERATION> public String getState()
        <CALLEE> Memento [17237886]       <CALLER> Originator [12386568]
```

Figure 9. Textual representation of the scenario diagram from the execution trace in Figure 8.

the classes whose objects are most likely to collaborate in a scenario diagram similar to the scenario diagram of a motif.

Indeed, considering that DeMIMA has a 100% recall [16], the static occurrences include all *potential* classes playing a role in a motif. Using these classes, the program source code, and its documentation, we manually build a scenario that exercises as many of these classes as possible and, thus, that is most likely to generate a trace including objects *indeed* playing a role in the motif.

For example, for the running example, DeMIMA provides static occurrences including the classes `Caretaker`, `Memento`, and `Originator`. Consequently, we develop a scenario that exercises all of these classes: in this simple case, this scenario consists solely of running the program once.

DeMIMA has an average precision of 34% and ranked occurrences from the most likely to the least likely using a weighting mechanism. Therefore, when building scenarios, developers should only consider the most likely occurrences and use their knowledge of the structure of the motif of interest to disregard occurrences that are too approximate.

Obtaining an exhaustive trace with complete coverage of the classes of interest may be impossible and executing the appropriate scenarios may require knowledge of the functionalities implemented using these classes. Also, it may happen that, due to feasibility constraints, only scenarios representing the most commonly used functionalities may be exercised. This problem is the dual problem of feature identification [2], which attempt to identify the source code implementing a feature available to users. We are not aware of any work tackling this problem and plan to study it in future work.

Yet, this problem does not limit the applicability of our approach. Indeed, given a set of n scenarios, MoDeC does not require all scenarios to identify the behavioural or creational motifs of interest. It only requires *one* scenario to increase our confidence that the motif is indeed present in the program because we can find both a static and a dynamic occurrence of the motif. If we cannot identify a dynamic occurrence of the motif using a scenario, then either the scenario is not adequate to identify the motif or the motif scenario diagram is too constraining. Only developers can decide whether to change the scenario or adapt the motif diagram in a continuous improvement cycle, using the explanations provided by the identification process, described in Section 5.

4.4. Instantiation of a Scenario Diagram

Figure 8 shows the trace obtained from the instrumented version of the running example. We analyse this trace to instantiate the corresponding scenario diagram. This instantiation process is independent of the target language of the program, as long as the execution trace follow the syntax in Figure 4. First, we identify the `sourceClassifier` of an operation by determining the callee of the start event immediately preceding the currently considered operation. In Figure 8, the preceding operation of `Message createMemento()` is `Message callCreateMemento()` at line 6. The `sourceClassifier` of `createMemento()` is then set as the `destinationClassifier` of `callCreateMemento()`. Once the `sourceClassifier` of each operation is identified in the execution trace, all needed data is available to instantiate a scenario diagram.

Figure 9 is a textual representation of the scenario diagram corresponding to the execution trace in Figure 8. For each event, a message of the appropriate type is instantiated. The component corresponding to the event currently considered in the execution trace is referred to as the *current component*. If the current component is of type `CombinedFragment`, we add the subsequent objects `Message` or `CombinedFragment` to its ordered list `operands` until the corresponding end event is met. Otherwise, they are added to the ordered list `components` of the `ScenarioDiagram` instance. Each time an object `Message` is instantiated, its corresponding `sourceClassifier` and `destinationClassifier` of type `Classifier` are also instantiated (if needed). The set `arguments` of a message is determined by processing the data positioned between the parenthesis of the corresponding event.

5. Identification of Motifs Scenario Diagrams in Programs Diagrams

Using the metamodel and techniques described in the previous sections, we can instantiate two scenario diagrams: one for a design motif and another for a program. Then, the identification of behavioural or creational motifs translates into a matching between the two scenario diagrams, *i.e.*, into finding all occurrences of the motif diagram in the program diagram.

We perform the matching between the two scenario diagrams using explanation-based constraint programming as in our previous work [16]. As shown in Figure 1, Step 3, we translate the identification into a CSP, which defines the variables, constraints, and domains representing the problem that an explanation-based constraint solver must solve to identify occurrences of the motifs. The CSP includes:

Variables. The set of variables `Classifier` and `Message` corresponds respectively to the entities `Classifier` and `Message` modelling the scenario diagram of a design motif.

Domains. The domains of each variable (`Classifier` or `Message`) corresponds to a set of integers, each corresponding to a unique `Classifier` or `Message` in the program diagram.

Constraints. The set of constraints among the variables corresponds to the relations among the entities of the scenario diagram defined by a motif.

For example, the Memento motif shown in Figure 2(a) is modelled by associating a variable with each of its entity: `var_createMemento`, `var_newMemento`, `var_setState`, `var_setMemento`, `var_getState`, `var_aCaretaker`, `var_anOriginator`, and `var_aMemento`. The domain of each variable corresponds to the entities in the scenario diagram of the program in which to identify occurrences of this motif. The scenario diagram modelling the Memento motif in Figure 9 includes ten operations: `main (String[])`, `<init>()`, `<init>()`, `callCreateMemento()`, `createMemento()`, `<init>()`, `setState()`, `undoOperation()`, `setMemento(Memento)`, and `getState()` as well as four objects: `ModelMementoTest`, `Caretaker`, `Originator`, and Memento. Therefore, the domains of the variables `var_aCaretaker`, `var_anOriginator`, and `var_aMemento` are of size 4 while the domains of variables `var_createMemento`, `var_newMemento`, `var_setState`, `var_setMemento`, and `var_getState` are of 10.

5.1. Constraints

We use binary constraints of the form `constraint(variable1, variable2)` to express the existence of relations between `variable1` and `variable2`. Constraints can be defined among variables of different types, `Classifier` and–or `Message`. They allow to describe motifs with precision because a relation can be expressed between `Message`s, `Classifier`s, or between a `Message` and a `Classifier`. To the best of our knowledge, in previous work, for example [31] or [16], constraints were only defined among variables of the same type.

We define the following set of constraints to express the relations among variables:

Constraint `calls(classifier1,message2)` (respectively `isCalled`) defines the relation *`classifier1` is the `sourceClassifier` of `message2`* (respectively `destinationClassifier`) between `classifier1` and `message2`. As shown in Algorithm 1, the domain of variables `classifier1` (respectively `message2`) corresponds to the instances of `Classifier` in the program scenario diagram (respectively `Message`). For each value taken by `message2`, there must be a corresponding value taken by `classifier1` so that `classifier1` is the `sourceClassifier` of `message2`. Conversely, for each possible value taken by `classifier1`, there must be a corresponding value taken by `message2` so that the `sourceClassifier` of `message2` is a `Classifier` in the domain of `classifier1`. Any value of `classifier1` and `message2` failing to satisfy this constraint is removed from the corresponding domain.

Constraint `creates(classifier1, message2)` (respectively `isCreated`) is similar to constraint `calls(classifier1, message2)`, except that `message2` is an instance of `Create` instead of `Operation`. For each possible value of `message2`, there must be a

Algorithm 1 Constraint `calls(classifier1, message2)`

```
1: toBeRemoved ← true
2: domain1 ← Domain(classifier1)
3: for i = 0 to Size(domain1) and toBeRemoved = true do
4:    listOfMessages ← MessagesWhoseCallerIs(classifier1)
5:    for j = 0 to Size(listOfMessages) and toBeRemoved = true do
6:       aMessage ← ElementAt(listOfMessages, j)
7:       domain2 ← Domain(message2)
8:       if ContainsMessage(domain2, message2) then
9:          toBeRemoved ← false
10:      end if
11:   end for
12: end for
13: toBeRemoved ← true
14: domain2 ← Domain(message2)
15: for i = 0 to Size(domain2) and toBeRemoved = true do
16:   domain1 ← Domain(classifier1)
17:   if ContainsClassifier(domain1, classifier1) then
18:      toBeRemoved ← false
19:   end if
20: end for
```

corresponding value of `classifier1` so that `classifier1` is an instance of `Create` and the `sourceClassifier` of `message2`.

Constraint `strictlyFollows(message1, message2)` defines the relation `message2` is executed after `message1`. The domains of variables `message1` and `message2` correspond to the instances of `Message` in the program diagram. For each possible value taken by `message2`, there must be a corresponding value taken by `message1` so that `message2` is called after `message1`. Conversely, for each possible value taken by `message1`, there must be a corresponding value taken by `message2` so that `message1` is called before `message2`. Any value of `message1` and `message2` failing to satisfy this constraint is removed from the corresponding domain.

Constraint `follows(message1, message2)` is similar to `strictlyFollows` in its definition but allows, in its satisfaction, other messages to occur between the two messages of interest, `message1` and `message2`. This constraint is useful when the implementation of `message1` may reasonably include intermediate calls before the actual call to `message2`.

Constraint `notEqual(classifier1, classfier2)` (respectively `(message1, message2)`) defines the relation `classifier` is not equal to `classifier2` (respectively `message1` is not equal to `message2`).

Constraint `parameterCalleeHasSameType(message1, message2)` defines that the first parameter of `message1` is of same type as the callee of `message2`. The domains of variables `message1` and `message2` correspond to the instances of `Message` in the program diagram. For each possible value taken by `message2`, there must be a corresponding value taken by `message1` so that the callee of `message2` is of the same type as the first parameter of `message1`. Conversely, for each possible value taken by `message1`, there must be a corresponding value taken by `message2` so that the first parameter of `message1` is of same type as the caller of `message2`. A value of `message1` or `message2` failing to satisfy this constraint is removed from the corresponding domain.

Constraint `isContainedIn(message1, message2)` defines the relation `message1` is eventually called during the execution of `message2`. The domains of variables `message1` and `message2` correspond to the instances of `Message` in the program diagram. For each possible value taken by `message2`, there must be a corresponding value taken by `message1` so that `message1` is called during the execution of `message2`, even if `message2` does not call *directly* `message1`. Conversely, for each possible value taken by `message1`, there must be a corresponding value taken by `message2` so that `message2` has not returned when `message1` is called. A value of `message1` or `message2` not satisfying this constraint is removed from the appropriate domain.

In the validation of our approach, these constraints were sufficient to express several design motifs. For example, the relations among the entities of the scenario diagram of the Memento motif in Figure 2(a) are translated one-to-one into the constraints shown in Figure 10.

```
1   strictlyFollows(var_createMemento, var_newMemento)
2   strictlyFollows(var_newMemento,var_setState)
3   strictlyFollows(var_setState, var_setMemento, 90)
4   strictlyFollows(var_setMemento, var_getState, 90)
5   calls(var_aCaretaker, var_createMemento)
6   isCalled(var_anOriginator, var_createMemento)
7   creates(var_anOriginator, var_newMemento)
8   isCreated(var_aMemento, var_newMemento)
9   calls(var_anOriginator, var_setState)
10  isCalled(var_aMemento, var_setState)
11  calls(var_aCaretaker, var_setMemento)
12  isCalled(var_anOriginator, var_setMemento)
13  calls(var_anOriginator, var_getState)
14  isCalled(var_aMemento, var_getState)
```

Figure 10. Constraints among the variables describing the Memento motif. By default, satisfying all constraints is mandatory; some constraints may be relaxed with a cost given as last parameter.

5.2. Solver

Complete occurrences, which are exactly similar to the scenario diagram of a design motif, may not be present in a program diagram, because the motif has been adapted by the developers to the program context. Approximate occurrences, which do not verify all the collaborations suggested by the motif, may be present and interesting for program comprehension. These occurrences correspond to solutions of a CSP that do not satisfy all the constraints.

We use an explanation-based constraint solver [23] to identify occurrences of the motifs. Given a CSP, an explanation-based constraint solver identifies automatically both complete and approximate occurrences. First, the solver iterates through each domain and removes the entities that satisfy all the constraints to form solutions representing complete occurrences of the motifs. It terminates its iterations on a contradiction: there are no more entities in the domains satisfying all the constraints. The solver then provides an explanation of the contradiction: the set of constraints that cannot be satisfied by the remaining entities. Relaxing or removing one or more constraints from the contradiction in the CSP allows the solver to form more solutions satisfying and, thus, to identify approximate occurrences. Relaxing or removing one of the constraints suggested in the explanation does not necessarily lead to a new solvable CSP, but the constraints can be relaxed and removed combinatorially until a solvable CSP is obtained or no constraint remains in the explanation.

Relaxation and removal may be performed manually or automatically. If performed manually, the user's is asked to choose the constraint(s) to relax or remove. To ease the user's choice, weights are given to the constraints that reflect an *a priori* hierarchy among them and allow the solver to present an ordered list of constraints from the explanation to the user. Thus, the user can restrict interactively the identification to the approximate occurrences of interest. Typically, the hierarchy among the constraints includes `strictlyFollows` as parent of `follows` because `follows` is less constraining: it allows other messages to occur between the two messages of interest.

```
1  var_createMemento = createMemento()
2  var_newMemento    = new Memento()
3  var_setState      = setState(String state)
4  var_setMemento    = setMemento()
5  var_getState      = getState()
6  var_aCaretaker    = Caretaker [14613018]
7  var_anOriginator  = Originator [12386568]
8  var_aMemento      = Memento [17237886]
```

Figure 11. Occurrence of the Memento motif.

For example, in Figure 10, two constraints can be relaxed, `strictlyFollows(var_setState, var_setMemento, 90)` and `strictlyFollows(var_setMemento, var_getState, 90)`, because their weights are 90%, *i.e.*, their satisfaction is not mandatory and their impact on the dynamic occurrence is to reduced our confidence by 10%. In this particular example, the two constraints could be relaxed into `follows` constraints.

Explanations, weights, and the hierarchy among constraints also allow automating the relaxation and removal process in the solver to generate automatically all complete and approximate occurrences without the user's input. For example, we apply the automated solver on the running example of the Memento motif and obtain the occurrence shown in Figure 11 without any manual intervention. We apply the automated solver in the validation of our approach to avoid any bias from the user's input.

An occurrence consists of a mapping between the CSP variables from the motif scenario diagram and the entities in the program scenario diagram. The entities can be operations, for example `createMemento()`, or objects, for example an instance of `Caretaker` with unique identifier 14613018.

6. Case, Empirical, and Comparative Studies

To validate our approach, we implemented MoDeC in a proof-of-concept implementation and now describe the identification of occurrences of the Command motif on the real-world program JHotDraw v5.4b1. This example highlights the use of DeMIMA occurrences to direct MoDeC and the use of MoDeC to increase the precision of DeMIMA. We also study empirically MoDeC on a set of five Java programs and report the precision and recall of MoDeC and discuss the threats to the validity of this empirical study. Finally, we compare the results of MoDeC with those of previous approaches.

6.1. JHotDraw Case Study

JHotDraw is a drawing editor that allows users to create and manipulate 2D vector figures. It has originally been developed in 1998 by Gamma and Eggenschwiler [11] as a show-case for the use of design patterns. It has since evolved into a full-fledged framework. We choose

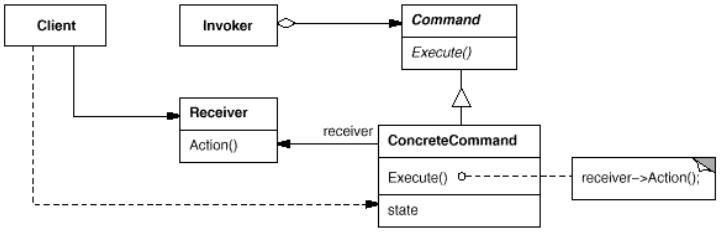


Figure 12. Description of the Command motif in term of its structure (from [12]).

```
1  inheritanceConstraint(var_ConcreteCommand, var_Command)
2  aggregationConstraint(var_Invoker, var_Command)
3  associationConstraint(var_ConcreteCommand, var_Receiver)
4  associationConstraint(var_Client, var_Receiver)
5  creationConstraint(var_Client, var_ConcreteCommand)
```

Figure 13. Constraint among the variables describing the Command motif in term of its structure.

JHotDraw because the motifs used in its implementation are well documented and, thus, can be used to validate the identified occurrences.

There are 18 occurrences of the Command motif in JHotDraw, all subclasses of interface `Command`. This information could be used in the descriptions of the motif either for DeMIMA or for MoDeC. However, we do not use this information in the following to respect the approaches described in [16] and in the previous sections.

We use the measures of precision and recall from the domain of information retrieval [10] to validate our approach. Precision assesses the number of true occurrences of a design motif identified in one scenario of a program, while recall assesses the number of true occurrences of a motif existing in one scenario of a program:

$$precision = \frac{|\{existing\ occurrences\ of\ a\ design\ motif\} \bigcap \{identified\ occurrences\ of\ a\ design\ motif\}|}{|\{identified\ occurrences\ of\ a\ design\ motif\}|}$$

$$recall = \frac{|\{existing\ occurrences\ of\ a\ design\ motif\} \bigcap \{identified\ occurrences\ of\ a\ design\ motif\}|}{|\{existing\ occurrences\ of\ a\ design\ motif\}|}$$

6.1.1. Identifying Static Occurrences with DeMIMA

We apply the approach described in [16] to describe and statically identify occurrences of the Command motif. The motif is described by modelling its structure using the constituents of the PADL meta-model, as shown in Figure 12. This structure is translated into a PADL model, which is turn is translated into a CSP. The CSP includes five variables: `var_Client`, `var_Command`, `var_ConcreteCommand`, `var_Invoker`, and `var_Receiver`. The

```
1  var_Invoker-1       = CH.ifa.draw.test.contrib.CommandMenuItemTest
2  var_Invoker-2       = CH.ifa.draw.util.CommandButton
3  var_Invoker-3       = CH.ifa.draw.standard.AbstractCommand.EventDispatcher
4  var_Invoker-4       = CH.ifa.draw.test.contrib.CommandCheckBoxMenuItemTest
5  var_Invoker-5       = CH.ifa.draw.standard.StandardDrawingView.DrawingViewKeyListener
6  var_Command         = CH.ifa.draw.util.Command
7  var_ConcreteCommand = CH.ifa.draw.standard.AlignCommand
8  var_Receiver        = CH.ifa.draw.framework.DrawingView
9  var_Client          = CH.ifa.draw.application.DrawApplication
```

Figure 14. An occurrence of the Command motif in term of its structure.

1. Launch JHotDraw.
2. Click on the rectangle tool.
3. Draw two rectangles.
4. Select the two rectangles.
5. Click on the align command.
6. Close JHotDraw.

Figure 15. A scenario to exercise the `AlignCommand` class in JHotDraw.

constraints among the variables, shown in Figure 13, describe the relationships among the classes in the class diagram, in Figure 12. They are systematically derived from the class diagram of the `Command` motif and therefore describe the inheritance, aggregation, associations, and creation among the classes.

Applying DeMIMA produces complete and approximate static occurrences of the motif: zero complete occurrences and 45 approximate occurrences[‡]. The approximate occurrences include the 18 true occurrences. Therefore, precision is equal to $18/45 = 0.4 = 40\%$ and recall to 100%, as expected when using DeMIMA. The approximate occurrences are ranked automatically according to their closeness to the original description of the motif.

For example, the first occurrences, closest to the motif, is shown in Figure 14. This occurrences is approximate because it does not satisfy all constraints, in particular the inheritance relationship between classes `Command` and `AlignCommand` is not direct. An abstract class `AbstractCommand` implements interface `Command` and is inherited by `AlignCommand` and all other commands. Also, this occurrence relates several possible invokers, with specific command, concrete command, receiver, and client. Associating more than one possible class with the invoker role reduces the computation time and the number of occurrences.

6.1.2. Building a Scenario from a Static Occurrence

The 45 occurrences of the Command motif include 365 distinct classes, for which it may be impractical to build scenarios. However, when considering, for example, the first five

[‡]The detailed 45 occurrences are available at `http://www.ptidej.net/downloads/experiments/jsme09/`.

```
notEqual(var_executeCallee, var_aCommand)
notEqual(var_aReceiver, var_aClient)
notEqual(var_aClient, var_aCommand)
notEqual(var_aCommand, var_anInvoker)
notEqual(var_aReceiver, var_aCommand)
strictlyFollows(var_newCommand, var_storeCommand)
strictlyFollows(var_storeCommand, var_execute)
strictlyFollows(var_execute, var_action)
creates(var_newCommand, var_aCommand)
isCreated(var_aClient, var_newCommand)
calls(var_aClient, var_storeCommand)
isCalled(var_storeCommand, var_anInvoker)
calls(var_anInvoker, var_execute)
isCalled(var_execute, var_executeCallee)
calls(var_aCommand, var_action)
isCalled(var_action, var_aReceiver)
parameterCalleeHasSameType(var_storeCommand, var_newCommand)
isContainedIn(var_action, var_execute)
```

Figure 16. Constraints among the variables describing the Command motif in terms of collaborations.

occurrences with highest ranking, which are most likely to be true occurrences, only 119 classes should be considered.

The first static occurrence includes 56 classes: 1 command, 17 concrete commands, 5 invokers, 27 receivers, and 26 clients. In particular, it includes the classes `CH.ifa.draw.util.Command` and `CH.ifa.draw.standard.AlignCommand` as the command and one concrete command, respectively. Therefore, using JHotDraw documentation, building a corresponding scenario is straightforward, given its features and simple user interface: we exercise the class playing the role of concrete command, class `AlignCommand` to exercise JHotDraw "align" feature. This scenario is shown in Figure 15.

It is possible that, with other programs, it would be more complex to build and exercise the appropriate scenario. In such a case, a unit test could also be implemented to exercise the appropriate classes. (Such a unit test exists with JHotDraw, we also used it as an alternative and did not find any difference in the following results.)

6.1.3. Identifying Dynamic Occurrences with MoDeC

We follow the steps shown in Figure 2:

Scenario Diagram of the Command Motif (Step 1). Figure 2(b), inspired by [12], describes the scenario diagram of the Command motif as collaborations among objects. A client that uses the Command motif creates a `Command` object, which will be executed by an `Invoker`. When a `Command` object is executed, an appropriate `Receiver` object performs the actual concrete action.

Scenario Diagram of the Program (Step 2). As described in Figure 1, Step 2.1, JHotDraw byte-code is the output from a Java compiler. Once all Java class files

```
var_newCommand    = <CREATE> public void <init>
                    (AlignCommand.Alignment newAlignment, DrawingEditor newDrawingEditor)
                    CALLEE AlignCommand [17652030] CALLER JavaDrawApp [31063377]
var_storeCommand  = <OPERATION> public synchronized void add (Command command)
                    CALLEE CommandMenu [5626173] CALLER JavaDrawApp [31063377]
var_execute       = <OPERATION> public void execute ()
                    CALLEE UndoableCommand [8175078] CALLER CommandMenu [5626173]
var_action        = <OPERATION> public Iterator drawingChangeListeners ()
                    CALLEE BouncingDrawing [5520561] CALLER AlignCommand [17652030]
var_aReceiver     = BouncingDrawing [5520561]
var_aClient       = JavaDrawApp [31063377]
var_aCommand      = AlignCommand [17652030]
var_anInvoker     = CommandMenu [5626173]
var_executeCallee = UndoableCommand [8175078]
```

Figure 17. An occurrence of the Command motif in term of its collaborations. (Package names have been removed for the sake of clarity.)

are instrumented (Step 2.2), the chosen scenario (Step 2.3) is executed (Step 2.4) to obtain a trace and, then, a scenario diagram of the program.

Resolving the CSP (Step 3). Following Figure 1, Step 3, we translate the description of the Command motif into a CSP. Each entity in the scenario diagram of Figure 2(b) is associated with a variable of the CSP. The set of constraints among the variables is shown in Figure 16 and is derived also directly from the scenario diagram.

Solving the CSP produces one dynamic occurrence of the Command motif, shown in Figure 17. According to the documentation of JHotDraw, the values of the variables in this occurrence indeed correspond to the objects and messages involved in the `Command` motif implemented for `AlignCommand` and other commands.

6.2. Accuracy Empirical Study

We follow the approach described and illustrated in the previous sections to obtain scenario diagrams of the Builder, Command, and Visitor design motifs, and of five Java programs: Dresden OCL v1.1, JHotDraw v5.4b1, JRefactory v2.6.24, PMD v1.8., and QuickUML 2001. We choose these programs because they include known occurrences of design motifs. These occurrences can be used as gold standards against which the accuracy of our approach can be compared.

Dresden OCL is a modular OCL (Object Constraint Language) toolkit that parses and type-checks OCL constraints and instruments Java code for runtime verification. It is also integrated into various CASE-tools and provides a SQL-code generator. JRefactory is a refactoring tool for the Java programming language that includes a pretty printer, a UML class diagram viewer, a coding standards checker, and computes program metrics. PMD is a Java source code analyser that finds unused variables, empty catch blocks, unnecessary object

creation, and so forth. Finally, QUICKUML is a class-diagram graphic editor that tightly integrates a core set of UML models.

Table I reports the precision and recall for the five programs, the three motifs, and DeMIMA and MoDeC. First, we applied DeMIMA on the two motifs, Command and Visitor, which we knew to be *not* implemented in DRESDEN OCL. As expected, DeMIMA has a 0% precision and 100% recall: it returns only false positive static occurrences. Using these false positive occurrences to build a scenario and using this scenario with MoDeC does not produce any true dynamic occurrences, and thus MoDeC has 100% precision and recall in these degenerate cases. We did not perform these computations for the other combinations of motifs/scenarios/programs and noted "–" such cases in the table.

Second, we applied the dynamic identification approach on a subset of scenarios/motifs of Table I, for which the corresponding motif is known *not* to be exercised by the scenario. Results show that no occurrences of the motifs were identified, as expected. For these evaluated scenarios, no false positives were found by MoDeC. Results appear as *N/E* in Table I.

Third, we observe that precision and recall are in general very high for each pair of scenarios/motifs with MoDeC. We explain these high precision and recall values by the fact that constraints can be relaxed to provide more (true) approximate occurrences when there are no more complete occurrences of a motif. Our experiments show that not all of the suggested constraints can be relaxed: only those that do not change the structure characterizing the motifs, nor the semantics of the original collaborations, should be relaxed/removed to obtain valid approximate occurrences. For the purpose of the evaluation, we manually limited the allowed approximations to those constraints that preserve the structure and the semantics of the motifs, *e.g.,* in the case of the Memento to those forms described in Section 5.

In conclusion, our approach works for several programs and various motifs, with good precision and perfect recall. Although the executed scenarios for each evaluated program are not an exhaustive list of all the possible scenarios, the current results are good predictors of the effectiveness of the proposed approach for the identification of behavioral and creational motifs using dynamic analysis. The increases of precisions *wrt.* DeMIMA range from 4× to more than 8×.

6.3. Comparative Study with Previous Approaches

We now compare the results of MoDeC with two previous approaches that also identify occurrences of the Command and Visitor in JHOTDRAW and JREFACTORY.

Similarity Scoring for Design Pattern Identification. Tsantalis *et al.* [40] introduce a measure of similarity among matrices representing either motifs or programs to provide better performance that previous static approaches and to deal with a wide range of structural motifs. The apply their approach on several motifs, including Command and Visitor, and on three programs, including JHOTDRAW v5.1 and JREFACTORY v2.6.24.

First, they combine the identification results of the Adapter and Command motifs because their static approach cannot distinguish between the two while MoDeC can because the scenario diagrams of the two motifs are different. Second, they only report classes playing some of the roles in each motif while MoDeC includes all the roles. Finally, they do not report the precision of their approach but the numbers of identified true positive occurrences. They

Programs	Scenarios	Design Motifs					
		Builder		Command		Visitor	
		DeMIMA	MoDeC	DeMIMA	MoDeC	DeMIMA	MoDeC
Dresden OCL	Transform OCL into SQL	10% 100%	100% (100%) 100%	0% 100%	100% (100%) 100%	0% 100%	100% (100%) 100%
JHotDraw	Cut and paste a rectangle	N/E	N/E	N/E	N/E	25% 100%	100% (50%) 100%
	Align figures	–	–	4% 100%	50% (33%) 100%	–	–
	Bring figures to front	–	–		100% (25%) 100%	–	–
	Send figures to back	–	–		100% (25%) 100%	–	–
	Group figures	–	–		100% (33%) 100%	–	–
JRefactory	Calculate a set of metrics of a class	–	–	–	–	50% 100%	84% (84%) 100%
PMD	Find variables with short names	–	–	–	–	25% 100%	100% (100%) 100%
QuickUML	Resize a diagram	N/E	N/E	N/E	N/E	100% 100%	100% (100%) 100%
	Enable toggle refresh from the menu	N/E	N/E	25% 100%	50% (33%) 100%	N/E	N/E
	Build a class from UML	29% 100%	100% (50%) 100%	N/E	N/E	N/E	N/E
	Average Precisions	19.5%	100% (75%)	9.6%	80% (29.8%)	40%	96% (84%)

Table I. Precision and recall calculated on scenarios and programs for different motifs using DeMIMA and MoDeC. In each row, the first line shows precision. For MoDeC, the first precision takes into account approximate occurrences of the motif while the precision in parenthesis includes only complete occurrences. The second line shows recall. (N/E means that a motif is "Not Exercised" by the scenario.)

also report a wrong occurrence of the Visitor motif in JHotDraw v5.1. In comparison, we report correct occurrences of both the Command and Visitor in JHotDraw v5.4b1.

Re-classification of the Design Patterns. Shi and Olsson [36] propose a re-classification of the design patterns in [12] to identify occurrences of their motifs in static models of programs. They use their re-classification to propose an identification approach, which they illustrate on four programs, including JHotDraw v6.0b1, and 17 motifs, including Visitor. First, they use a static model of programs to identify occurrences of behavioural, creational, and structural motifs—which they re-classify as language-provided, structure-driven, behaviour-driven, domain-specific, and generic-concepts motifs—while we use a dynamic analysis. Second, they report more data than any other identification approach, including delegation and method forwarding (propagation), for example. According to their on-line results[§], they do not identify any occurrence of the Visitor in JHotDraw while its source code hints at one occurrence, implemented by interface `FigureVisitor` and classes `DeleteFromDrawingVisitor` and `InsertIntoDrawingVisitor`, since v5.4b1. MoDeC identifies correctly this occurrence.

We conclude that previous works proposed invaluable advances to the state-of-the-art. MoDeC builds on these works to further improve the precision of the identification of behavioural and creational motifs.

6.4. Threats to Validity

The previous case, empirical, and comparative studies support our claim that our dynamic identification approach for behavioural and creational design motifs has a good precision and recall and that its results are better than those of a static approach and of previous works. However, the following threat to their validity should be mitigated in the future.

Construct validity threats concern the relation between theory and observation; in this paper, they are mainly due to errors introduced in measurements. The count of true positive occurrences of the identified motifs has been performed manually and it is possible that some of these occurrences are not true positive while some true positives were missed. We reduced this threat by asking other experts in software engineering to validate the occurrences.

Internal validity threats do not affect our case and empirical studies, being exploratory studies [44]. Thus, we cannot claim causation, but can only state that on our data (1) MoDeC seems to increase good precision with respect to static approaches, (2) using static occurrences to produce scenarios seems to provide good results, and (3) MoDeC seems to have a higher precision and recall than previous approaches.

External validity threats concern the possibility to generalise our results. First, we are aware that our study has been performed on limited numbers of programs and motifs, thus generalisation will require further empirical studies. Yet, JHotDraw has been used in many previous studies in design pattern identification and we also used three other motifs and four other programs to illustrate our approach. Different motifs and programs could have led to different results and should be studied in future work. However, within their limits, our

[§] `http://www.cs.ucdavis.edu/~shini/research/pinot/results/jhotdraw_results.html`, last visited on May 5, 2009.

case, empirical, and comparative studies confirm our claim and previous works on dynamic identification approaches.

Conclusion validity threats concern the relationship between a treatment and its outcome: whether or not it is reasonable to believe that one affects the other. We believe that these threats do not affect our studies because it is reasonable to believe that it is the combination of dynamic and static occurrences that lead to the observed precision and recall.

Reliability validity threats concern the possibility of replicating this study. We attempted here to provide all the necessary steps to duplicate our approach and replicate the case, empirical, and comparative studies. Moreover, all the programs and data used in the studies, *e.g.*, scenarios, are freely available on the Internet. We provide some data on-line[‡] and will gladly provide our implementation upon request.

7. Discussion

Our approach must be considered in the light of the following discussions. First, users of our approach need not be experts in either design patterns or the program under analysis. Indeed, we provide a library of design motif descriptions that is reusable in any context. Moreover, the use of explanation-based constraint programming provide explanations about the identified occurrences, thus easing their use by users while allowing to fine-tune their descriptions. In addition, the following discussions must be considered:

7.1. Choice of the Motifs

Our approach cannot be fully automated because the description of motifs and the building of scenarios to reverse-engineer traces of a program are intrinsically manual, as shown in Figure 1, because, to the best of our knowledge, (1) there do not exist general and available models of design motifs from which to obtain their description and (2) the identification of the feature exercising particular classes has not been investigated so far, as the dual problem of feature identification [2]. Consequently, our approach should be applied to behavioural and creational motifs with limited structural characteristics, *i.e.*, these motifs that have a low precision in DeMIMA and other structural design pattern identification approaches.

7.2. Description of the Motifs

As explained in Section 5, we describe motifs in terms of collaborations among objects, as given in [12]. However, as classes participating in a motif need not be collaborating precisely according to the proposed collaborations, a user interface could help users to easily describe the collaborations among objects and generate the scenario diagram and–or CSP as well as grasp the impact of relaxing or removing such or such constraints.

7.3. Choice of the Programs

Although the programs are intentionally designed to have clear implementations of the motifs in [12] with explicit class and method names, they still are a good benchmark for the evaluation of our approach, because they are well documented and have already been the subjects of evaluation in previous works.

It is useful to evaluate programs for which the presence of a motif is known, because the measures of precision and recall are calculated with the number of existing occurrences of the motif. Also, the proposed identification approach is independent of the class and method names. Therefore, this choice of programs does not constitute directly a threat to the generalisability of MoDeC. Yet, we plan to evaluate our approach on more programs in the future to confirm its generalisation.

The evaluated programs are all in Java. As previously discussed in Section 4, the target language influences only the collection of execution traces. It does not affect the identification of the motifs. Furthermore, the principles that we proposed in Section 4.2 to instrument methods can be generalised to most object-oriented programming languages.

7.4. Choice of the Scenarios

While evaluating our approach, we had to choose the scenarios to be executed. An ideal evaluation requires to trigger as many scenarios as possible, to evaluate the approach on all of the functionalities of a program. However, due to feasibility constraints, we only used a subset of the scenarios representing the most commonly used functionalities, which descriptions are generally available in the documentation. Future work includes the merging of several scenario diagrams to obtain *sequence diagrams*.

7.5. Scalability and Performance

One of the key challenges while using dynamic analysis to monitor the behavior of a program is the large amount of generated data. As the size of the program grows, the execution trace also grows, and execution time required to solve the CSP increases. For instance, as of today, the CSP solver needs approximately 24 hours to solve the identification problem for an execution trace in which 4,000 messages were traced, 9 variables, and 14 constraints (excluding the combination of the two possible soft constraints).

We did not try to improve the performance of our proof-of-concept implementation in time or space because in this paper we only wanted to show that our approach increases precision and recall. The current implementation combines freely available tools, in particular a generic-purpose explanation-based constraint solver. This solver as well as the definition of the constraints should be optimised to cope more effectively with larger execution traces.

Among the most commonly used abstractions to cope with high volume of dynamic data, we used start and end markers to specify respectively the beginning and end of exercising a feature. We thus placed two markers in the execution trace of the scenario to specify the beginning and end of a feature, for example the *align* feature of JHotDraw, which we believed would exercise the classes of interest participating in occurrences identified by DeMIMA. Operations

outside each a start and end markers can be omitted from the execution trace. Results after applying our identification approach both on the original and the shorten execution trace return identical occurrences. The marker mechanism reduces the volume of dynamic data but still needs some more refinement to assure that no occurrences of a motif are omitted because some method executions are eliminated from the original execution trace.

We will also consider summarising an execution trace to extract only its main content. Such strategy removes implementation details such as calls to utility methods from the trace, which have no obvious use for the identification of behavioral and creational motifs. Some works have already researched this direction, for example [19, 20], and we plan to reuse such an approach to further decrease the size of the trace, and thus improve performance. When summarising traces, recall could decrease if data useful to the identification is removed. Although we are confident that existing approaches took care of not removing *too much* data, empirical studies will be performed to assess this possible decrease.

7.6. Comparing and Filtering Static and Dynamic Occurrences

In addition to using static occurrences to build scenarios, we can also compare and filter static occurrences using dynamic occurrences to obtain more precise sets of static occurrences without excluding true occurrences.

Comparing Occurrences. The static and dynamic occurrences do not provide the same data. Differences exist because static occurrences can only include the *declaring* types of methods and relationships and the *declared* types of method calls, while the dynamic occurrences include the *concrete* types of the collaborating objects. For example, the static occurrence of Command in JHotDraw, shown in Figure 14, identifies class `DrawingApplication` as receiver while the dynamic occurrence, shown in Figure 17, identifies class `JavaDrawApp`. Class `JavaDrawApp` is actually a sub-type of class `DrawingApplication`. Thus, dynamic occurrences are, as expected, more precise than static occurrences with respect to types. Consequently, dynamic occurrences can make the static occurrences more precise *wrt.* types, without impacting the overall precision and recall of the static approach.

Filtering Occurrences. Dynamic occurrences can also be used to filter static occurrences to obtain a new set of occurrences, whose precision is equal to or greater than the precisions of the original set of static occurrences. Indeed, dynamic occurrences can *confirm* static occurrences: for example, we can iterate through the 45 static occurrences of Command in JHotDraw and exclude occurrences in which the Client role is not played by class `DrawingApplication` or one of its subclass. We obtain a set of 93 occurrences. Therefore, we increase the precision from 4% to $18/93 = 0.19 = 19\%$ while keeping the recall equal to 100%. We could use other role to increase the precision but in this particular case, other roles would not yield to another increase of precision because no further roles in the static occurrences match the remaining roles in the dynamic occurrences.

Combining static and dynamic occurrences can lead to an increase in precision from 4% to 19%. Despite interesting improvement, the precision is still low for the following reason.

The description of the motif in DeMIMA leads to large number of false positives due to the behavioural nature of the motif and the static nature of the approach. Therefore, many false positives must be filtered out by MoDeC. Yet, the description of the motif in MoDeC is not manually optimised to increase precision by taking into account particular cases but follows systematically the scenario diagram given in the Collaborations section of Gamma *et al.*'s book to ensure 100% recall. Therefore, MoDeC is able to filter out some false positives but not all of them, because some of them would require to fine-tune the description of the motif in MoDeC. In the future, we plan to further refine the descriptions to address particular variants of design motifs that would lead to a greater improvement in precision. Refined descriptions could include negative constrains as suggested by Heuzeroth [21].

8. Related Work

The identification of motifs in object-oriented programs has been the subject of many works. In particular, the identification of structural motifs has been investigated since as early as 1998 [43] by many authors. The identification of behavioural and creational design motifs using dynamic analysis is the subject of fewer work.

8.1. Structural Motif Identification using Static Analysis

Since their inception in 1995, design patterns have been the subject of many works related to their detection in programs. Most of these works use static data. For example, Wuyts [43] published a precursor work on structural motif identification. His approach consisted of representing programs as Prolog facts and in describing motifs as predicates on these facts. Facts were extracted using static analysis. This approach had performance issues due to the use of a Prolog engine. It could not deal with variants automatically. It had limited precision and recall according to subsequent studies. It was followed by many other works to overcome its limits. These works use different data and different representation and detection techniques. For example, Quilici *et al.* [31] introduced the use of constraint programming to describe motifs as constraint satisfaction problems, while improving both the descriptions and the performance. Guéhéneuc *et al.* [18] drew inspiration from these works and introduced the use of explanation-based constraint programming and of a dedicated metamodel to describe both motifs and programs, including binary class relationships [15] to improve both the representation and the handling of variants. Kniesel *et al.* [25] proposed and evaluated the use of a transformation framework to specify and identify motifs with good precision and performance. Many other works exist but, to the best of our knowledge, none of these previous works focused on behavioral and creational motifs.

8.2. Structural Motif Identification using Static and Dynamic Analyses

Some design patterns are categorised as creation or behavioural but display a strong structure. For example, the Observer motif is characterised both by its structure (as illustrated in [12]) and the behaviour of its objects at runtime.

Heuzeroth *et al.* [22] proposed an approach that uses both static and dynamic data to identify interaction "patterns" and illustrated their approach on the Observer motif using a dedicated detection algorithm. It is unclear how their approach can be generalised to pure-behavioural/creational motifs. Shawky *et al.* [35] proposed a similar approach to improve the precision and recall of a static identification approach with limited benefits.

Some previous work also used dynamic data in addition to structural data to improve precision and recall. In particular, most previous works on the identification of structural motifs use data related to method calls, which can be considered as dynamic data, for example [1] or [15] used in [18].

8.3. Creational and Behavioural Motif Identification using Dynamic Analysis

Few authors have tackled the problem of specifying and identifying behavioural and creational motifs using dynamic analysis. Following Heuzeroth *et al.* [21], Wendehals [41] also proposed to combine static and dynamic analyses to allow identifying behavioural and creational motifs. They defined an approach in which occurrences are identified using static and dynamic analyses in parallel. Identified static and dynamic occurrences are presented to the maintainers for acceptance or rejection.

Wendehals and Orso [42] presented an approach to specify behavioural and creational motifs using UML 2.0 sequence diagrams and to use static occurrences as input of the dynamic analysis. Sequence diagrams are first customised by the maintainers and then converted into finite automata for the identification. Static occurrences are weighted with confidence values based on the results of the dynamic analysis.

Wendehals' work has been a source of inspiration for our work. We improve on this previous work by using the static occurrences to direct the generation of the trace and thus increase the likelihood to capture the interactions among objects participating in the motifs. We also introduce the use of explanation-based constraint programming to avoid having to customise the motifs. Finally, we present the results of our approach on a greater number of motifs.

8.4. Recovery of Sequence Diagrams

The recovery of sequence diagrams has been tackled by several authors. An important contribution to this field is the work of De Pauw *et al.* [29], which describes a model to visualize data about the execution of object-oriented programs. Briand *et al.* [5] proposed a method to reverse engineer UML sequence diagrams from execution traces. They used the recovered traces and a metamodel to describe UML v1.x sequence diagrams. Rountev *et al.* [32] described a first algorithm to reverse engineer UML v2.0 sequence diagrams by control-flow analysis. Their approach did not consider data obtained by dynamic analysis, and thus

is limited by the accuracy of the control-flow analysis. Briand *et al.* [4] introduced a complete approach to recover scenario diagrams using execution traces.

9. Conclusion

We proposed MoDeC, a 3-step approach to identify behavioral and creational design motifs in source code of programs using dynamic analysis. To reduce the number of scenarios, we used a static approach to design pattern detection. We described behavioral and creational motifs as scenario diagrams. We showed the reverse-engineering of scenario diagrams of a given program by means of dynamic analysis through intermediate-code instrumentation. Finally, we implemented the concrete identification using explanation-based constraint programming, to identify, in the scenario diagrams of a program, objects and messages satisfying the set of constraints derived from the scenario diagram of a motif. We illustrated our approach on the real-world program JHotDraw and the Command motif and showed that combining MoDeC with our previous static approach, DeMIMA, leads to an increase in precision from 4% to 19% while keeping the recall equals to 100%. We also evaluated our approach on Dresden OCL Toolkit, JHotDraw, JRefactory, PMD, and QuickUML with the Builder, Command, and Visitor motifs to show its good precision and perfect recall.

Future work includes evaluating our approach on more motifs and programs; merging scenario diagrams to obtain sequence diagrams; using abstraction and summary mechanisms to reduce the size of execution traces without losing relevant data; adding new constraints and improving the CSP of the motifs to obtain higher precision without impacting recall. It also includes improving the performance in space and time of the current implementation; analysing source code to suggest which feature to exercise to trace classes and objects of interest, obtained from DeMIMA; increasing further precision by tailoring motif descriptions.

REFERENCES

1. Giuliano Antoniol, Gerardo Casazza, Massimiliano di Penta, and Roberto Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59:181–196, November 2001.
2. Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: An epidemiological metaphor. *Transactions on Software Engineering (TSE)*, 32(9):627–641, September 2006. 15 pages.
3. Apache Jakarta Project. *Byte Code Engineering Library*, June 2006.
4. Lionel Briand, Yvan Labiche, and Johanne Leduc. Towards the reverse engineering of UML sequence diagrams for distributed Java software. *Transactions on Software Engineering*, 32(9), September 2006.
5. Lionel Briand, Yvan Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. *Proceedings of the* 10^{th} *Working Conference on Reverse Engineering*, pages 57–66, November 2003.
6. Julio Cesar Sampaio do Prado Leite, Jorge Horacio Doorn, Graciela D. S. Hadad, and Gladys N. Kaplan. Scenario inspections. *Requirements Engineering*, 10(1):1–21, January 2005.
7. Julio Cesar Sampaio do Prado Leite, Graciela D. S. Hadad, Jorge Horacio Doorn, and Gladys N. Kaplan. A scenario construction process. *Requirements Engineering*, 5(1):38–61, July 2000.
8. Alexander Egyed. A scenario-driven approach to trace dependency analysis. *Transactions on Software Engineering*, 29(2):1–17, February 2003.
9. Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.
10. William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, facsimile edition edition, June 1992.

11. Erich Gamma and Thomas Eggenschwiler. JHotDraw. Web site, 1998. members.pingnet.ch/gamma/JHD-5.1.zip.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.
13. Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.
14. Andreas Gregoriades and Alistair Sutcliffe. Scenario-based assessment of nonfunctional requirements. *Transactions on Software Engineering*, 31(5):392–409, May 2005.
15. Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In Doug C. Schmidt, editor, *Proceedings of the* 19^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301–314. ACM Press, October 2004. 14 pages.
16. Yann-Gaël Guéhéneuc and Giuliano Antoniol. DeMIMA: A multi-layered framework for design pattern identification. *Transactions on Software Engineering (TSE)*, 34(5):667–684, September 2008. 18 pages.
17. Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine – A tool for dynamic analysis of Java programs. In Wolfgang Emmerich and Dave Wile, editors, *Proceedings of the* 17^{th} *Conference on Automated Software Engineering (ASE)*, pages 117–126. IEEE Computer Society Press, September 2002. 10 pages.
18. Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design-patterns identification. In Christian Bessière, editor, *Proceedings of the* 1^{st} *IJCAI Workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001. 8 pages.
19. Abdelwahab Hamou-Lhadj, Edna Braun, Daniel Amyot, and Timothy Lethbridge. Recovering behavioral design models from execution traces. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 112–121, Washington, DC, USA, 2005. IEEE Computer Society.
20. Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society.
21. Dirk Heuzeroth, Thomas Holl, and Welf Löwe. Combining static and dynamic analyses to detect interaction patterns. In Hartmut Ehrig, Bernd J. Krämer, and Atila Ertas, editors, *Proceedings the* 6^{th} *world conference on Integrated Design and Process Technology*. Society for Design and Process Science, June 2002.
22. Dirk Heuzeroth, Welf Löwe, and Stefan Mandel. Generating design pattern detectors from pattern specifications. In *18th IEEE International Conference on Automated Software Engineering (ASE) 2003*. IEEE, 2003.
23. Narendra Jussien. e-Constraints: Explanation-based constraint programming. In Barry O'Sullivan and Eugene Freuder, editors, 1^{st} *CP workshop on User-Interaction in Constraint Satisfaction*, December 2001.
24. Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In David Garlan and Jeff Kramer, editors, *Proceedings of the* 21^{st} *International Conference on Software Engineering*, pages 226–235. ACM Press, May 1999.
25. Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In Tom Tourwé, Andy Kellens, Mariano Ceccato, David Shepherd, and Marius Marin, editors, *Proceedings of the* 3^{rd} *AOSD Workshop on Linking Aspect Technology and Evolution*. ACM Press, March 2007.
26. Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In Linda M. Wills and Ira Baxter, editors, *Proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*, pages 208–215. IEEE Computer Society Press, November 1996.
27. Jörg Niere. Fuzzy logic based interactive recovery of software design. Presented at the ICSE Doctoral Symposium, May 2002.
28. Object Management Group. *UML 2.0 Superstructure Specification*, October 2004.
29. Wim De Pauw, Doug Kimelman, and John M. Vlissides. Modeling object-oriented program execution. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, volume 821, pages 163–182. Springer-Verlag, July 1994.
30. Niklas Pettersson and Welf Löwe. Efficient and accurate software pattern detection. In Pankaj Jalote, editor, *Proceedings of the* 13^{th} *Asia Pacific Software Engineering Conference*, pages 317–326. IEEE Computer Society Press, December 2006.

31. Alex Quilici, Quing Yang, and Steven Woods. Applying plan recognition algorithms to program understanding. *journal of Automated Software Engineering*, 5(3):347–372, July 1997.
32. Atanas Rountev, Olga Volgin, and Miriam Reddoch. Static control-flow analysis for reverse engineering of UML sequence diagrams. *Proceedings of the* 6^{th} *Workshop on Program Analysis for Software Tools and Engineering*, pages 96–102, September 2005.
33. Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley Professional, 1^{st} edition, February 2003.
34. Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of Java software. In Bill Scherlis, editor, *Proceedings of* 5^{th} *international symposium on Foundations of Software Engineering*, pages 10–16. ACM Press, November 1998.
35. Doaa M. Shawky, Salwa K. Abd-El-Hafiz, and Abdel-Latif El-Sedeek. A dynamic approach for the identification of object-oriented design patterns. *Proceedings of the* 2^{nd} *International Conference on Software Engineering*, pages 138–143, February 2005.
36. Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In Sebastian Uchitel and Steve Easterbrook, editors, *Proceedings of the* 21^{st} *International Conference on Automated Software Engineering*, pages 123–134. IEEE Computer Society Press and ACM Press, September 2006.
37. Janice Ka-Yee Ng and Yann-Gaël Guéhéneuc. Identification of behavioral and creational design patterns through dynamic analysis. In Andy Zaidman, Abdelwahab Hamou-Lhadj, and Orla Greevy, editors, *Proceedings of the* 3^{rd} *International Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 34–42. Delft University of Technology, October 2007. TUD-SERG-2007-022. 9 pages.
38. Alistair G. Sutcliffe, Neil A. M. Maiden, Shailey Minocha, and Darrel Manuel. Supporting scenario-based requirements engineering. *Transactions on Software Engineering*, 24(12):1072–1088, December 1998.
39. Dave Thomas. Reflective software engineering – From MOPS to AOSD. *Journal of Object Technology*, 1(4):17–26, September 2002.
40. Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros Halkidis. Design pattern detection using similarity scoring. *Transactions on Software Engineering*, 32(11), November 2006.
41. Lothar Wendehals. Improving design pattern instance recognition by dynamic analysis. In Jonathan E. Cook and Michael D. Ernst, editors, *Proceedings of the* 1^{st} *ICSE Workshop on Dynamic Analysis*. IEEE Computer Society Press, May 2003.
42. Lothar Wendehals and Alessandro Orso. Recognizing behavioral patterns at runtime using finite automata. In Neelam Gupta and Andy Podgurski, editors, *Proceedings of the* 4^{th} *ICSE Workshop on Dynamic Analysis*. ACM Press, May 2006.
43. Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *Proceedings of the* 26^{th} *Conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.
44. R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London, 2002.