Towards Abstract Interpretation for Recovering Design Information

Coen De Roover, Kris Gybels [1], Theo D'Hondt

Programming Technology Lab
Vrije Universiteit Brussel
Brussels, Belgium
{*cderoove, kris.gybels, tjdhondt*}*@vub.ac.be*

Abstract

It is a well-known problem that design information of object-oriented programs is often lost or is not kept up-to-date when the program evolves. This design information can be recovered from the program using such techniques as logic meta programming. In this technique logic queries are used to check whether the program is implemented along certain well-known patterns. Currently the technique relies on structural information and patterns are expressed in the queries as conditions over structural elements of the program. Some patterns are however better expressed in dynamic terms which requires behavioural information about the program. Such information can be obtained from execution traces of the program, but these record only one possible input dependent program execution out of many. Abstract interpretation of the object-oriented program could provide a well-founded means for extracting the necessary behavioural information.

Key words: logic meta programming, design recovery, dynamic analysis, abstract interpretation

1 Introduction

It is a well-known problem that design information of programs is often lost or is not kept up-to-date when the program evolves [11]. This makes later maintenance of the program difficult as such maintenance usually requires the programmer to have an understanding of the global structure of the program: the relationships between modules, or classes in the case of object-oriented

[1] Kris Gybels is a research assistant of the Fund for Scientific Research, Flanders, Belgium (F.W.O.)

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

programming. One goal of reverse engineering research is to provide techniques for aiding the programmer in recovering such design information from the program itself.

One technique that can be used for recovering design information is Logic Meta Programming [11]. This technique revolves around the use of logic programming for posing queries on a program. Using a library of logic rules that define which high-level relationships between program elements hold under which conditions, a programmer can build an understanding of a program by querying it for such relationships.

One particularly interesting kind of design information about an object-oriented program is where it makes use of design patterns [4]. These define patterns for using classes in a certain way to solve common problems, such as the Strategy pattern for dynamically changing an object's behaviour and the Visitor pattern for performing operations on all objects in a tree structure. The fact that a program makes use of these patterns is usually not immediately obvious from the code because the pattern is not a single element in the program but rather is defined by structural relationships between elements as well as the behaviour that arises from it. It has been shown that knowledge about the presence of these patterns improves the software maintenance process [9].

While Logic Meta Programming (LMP [2]) can currently already be used to detect design patterns based on detecting the salient structural relations defining a pattern [2], there is no model yet for detecting patterns based on the behaviour they give rise to. Such a model is necessary as in some cases a pattern is more easily detectable based on the behaviour, or because the resulting logic rules become more readable because they better capture the pattern.

We have currently explored how pattern detection rules based on behaviour can be written using an ad-hoc model based on execution trace information. Our position for this workshop is that Abstract Interpretation research can provide a better theoretically-founded model. We wish to receive further feedback from the Abstract Interpretation community on this position. To provide the necessary background this paper further explains the following: section 2 gives a brief overview of the current LMP approach, section 3 describes a particular design pattern in more detail, sections 4 and 5 respectively contrast the current structural-based pattern detection approach and our proposed novel behavioural-based one and in section 6 we consider the question of how abstract interpretation can be used.

[2] In the remainder of this paper, the acronym LMP will denote Logic Meta Programming.

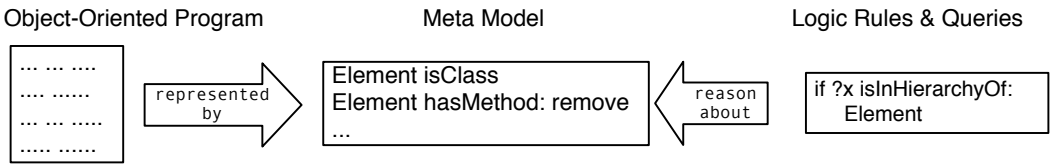

Fig. 1. An overview of Logic Meta Programming with a structural meta model

2 Logic Meta Programming

Logic Meta Programming is a technique in which a Prolog-like language [3] is used as a meta language for reasoning about object-oriented programs. Over the years, it has been applied to a variety of problems in object-oriented software engineering, some examples are: reasoning about object-oriented design [11,10]; checking and enforcing programming patterns [7]; supporting evolution of software applications [8] and checking architectural model conformance [12]. Following the example of these researchers, we use the SOUL logic meta programming system to conduct experiments on programs written in Smalltalk [5]. The SOUL approach to logic meta programming is however generic and can be applied to most class-based object-oriented programming languages, as is evidenced by the existence of SOUL for Java [2].

Figure 1 illustrates the overall approach of LMP. To allow the use of logic queries to reason about the program, it is represented as logic facts according to a meta model. In the current structural meta model these facts state the classes and methods present in the program, and the basic relations between these such as one class being the subclass of another.

Starting from the basic structural facts, more complex relationships can be derived by defining the appropriate logic rules. For example the following rules express what it means for one class to (in)directly be a subclass of another [3]:

```
?directSubclass isInHierarchyOf: ?root if
  ?directSubclass isSubclassOf: ?root.

?indirectSubclass isInHierarchyOf: ?root if
  ?indirectSubclass isSubclassOf: ?parent,
  ?parent isInHierarchyOf: ?root
```

The first rule expresses that one class is in the class hierarchy of another class when it is the subclass of that class. The second rule expresses that classes that are the subclasses of a class that is in the hierarchy of some class are also in the hierarchy of that same class.

As with regular Prolog, one can use the `isInHierarchyOf:` predicate in logic queries both to *verify* whether there is a hierarchy relationship between two classes and to *detect* the classes another class has in its hierarchy. In the first of the two example logic queries below, the `isInHierarchyOf:` predicate

[3] The logic language used for LMP has a syntax that is somewhat different from Prolog's [6]: an expression such as `?a m: ?b n: ?c` is a functor with a name consisting of the two words 'm' and 'n'; `?a`, `?b` and `?c` are variables.

is used to verify that the class `String` somehow inherits from the class `Object`, in the second query it is used to find all classes that inherit (in)directly from the class `Object`. For the first query the logic evaluator will try to logically prove that the `isInHierarchyOf:` predicate holds for the given arguments and for the second query it will return all the values for the variable `?x` that make the predicate hold.

```
if String isInHierarchyOf: Object
if ?x isInHierarchyOf: Object
```

Thus for better understanding a program, a programmer can use LMP to verify her intuitions about or detect the relationships between program elements using logic rules. While even simple rules like `isInHierarchyOf:` can aid in this understanding, the conditions for detecting more interesting and complex relationships can be encoded in rules as well as will be discussed in the next section for the Visitor design pattern.

3 Design Patterns

In the following section we will demonstrate how logic meta programming can be used to detect and verify complex software patterns. To this end, we will first describe a prototypical software pattern which we will use as a running example: the Visitor design pattern. We will particularly emphasise and compare the two complementary perspectives from which this software pattern can be described: a structural perspective and a behavioural one. As will be explained later, detecting the Visitor design pattern on the basis of the behavioural perspective is more reliable but requires a new approach to LMP.

From the structural point of view, a pattern is described in terms of class hierarchies and specific statements in methods. The behavioural point of view describes a pattern in contrast by the protocol that governs the interaction between the run-time entities the pattern is composed of.

The Visitor design pattern is a pattern of moderate complexity and makes a perfect case study for the merits of the inclusion of behavioural information for the detection of software patterns. It is one of the twenty-three design patterns introduced by the "gang of four" in their book [4] on reusing proven and often re-occurring software designs.

The pattern solves a common problem where many unrelated operations need to be performed on objects of different types held together in a compound structure. Its essence is a well-defined protocol between the class traversing a compound structure and the components it contains. The protocol demands that a visited component notifies the traversing class of its type. This allows operations relying on a traversal of the compound structure to be defined separately from the components themselves.

A more detailed description of the Visitor protocol above can be undertaken either from the structural or the behavioural point of view. From the

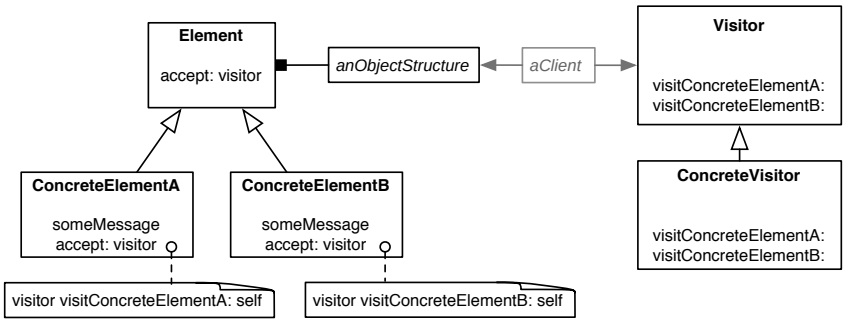

Fig. 2. Class diagram describing the architecture of the Visitor design pattern

former, we are mainly interested in the implementation of this protocol. The architectural building blocks are shown in figure 2. The `Visitor` abstract class has a method `visitConcreteElementX:` for each element of type `X` in the object structure. Instead of scattering the implementation of the compound traversal operation across the entire object structure, the corresponding partial implementations can be gathered into `visitConcreteElementX:` methods of `Visitor` subclasses. Each component in the compound object structure must in turn implement an `accept:` method.

From the behavioural point of view, the objects in the object structure accept a `Visitor` subclass with their `accept:` method and subsequently call the `visitConcreteElementX:` method corresponding with their type on the received visitor. More specifically, if we study the dynamic behaviour of the Visitor Design Pattern using the annotated sequence diagram shown in figure 3, we can conclude that a *recursive double dispatching* over instances held by a parent node characterises this pattern's behaviour. The visitation of `anObjectStructure` begins and ends at certain moments in time between which a visitation of the subelements `aConcreteElementB` and `aConcreteElementA` occurs. The latter visitation comprises a third visitation on `aConcreteElementC`.

4 Detection of the Visitor Structure

The current approach to logic meta programming in SOUL uses a static meta model for representing a program's source code as logic predicates. This kind of meta model agrees with the structural point of view from which the Visitor design pattern can be described. It results in a straightforward translation of the structural architecture shown in figure 2 into an equivalent executable logic rule shown in figure 4.

The `withSelector:visits:withSelector:` predicate states that a `?visitor` with method selector `?visitSelector` visits a `?element` class with method selector `?accept`. This logic statement is true if all of the following conditions are satisfied. First of all, the `?visitor` variable must be bound to a class from the program's source code. This class must furthermore implement a

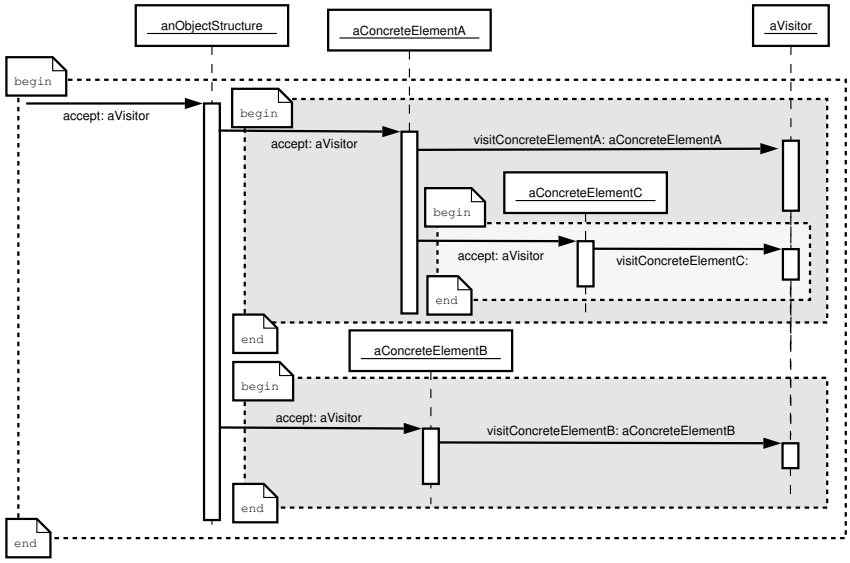

Fig. 3. An annotated sequence diagram demonstrating the recursive nature of the Visitor design pattern

```
1  ?visitor withSelector: ?visitSelector visits: ?element  withSelector: ?accept if
2      ?visitor isClass,
3      ?visitor implements: ?visitSelector,
4      ?element isClass,
5      ?element implements: ?accept withBody: ?acceptBody,
6      ?acceptBody methodArguments: ?acceptArgs,
7      ?acceptBody methodStatements:
8          <return(send(?visitor, ?visitSelector,?visitArgs ))> ,
9      ?visitArgs contains: variable([#self]),
10     ?acceptArgs contains: ?visitor.
```

Fig. 4. Structural Visitor detection rule

```
1  ?visitor visits: ?composite from: ?begin till: ?end invokedBy: ?invoker if
2    ?invoker doubleDispatchesOn: ?composite
3             selector: ?acceptselector
4             at: ?begin
5             andOn: ?visitor
6             selector: ?visitselector
7             at: ?end,
8    (?visitor visits: ?part from: ? till: ? invokedBy: ?)
9      forall: (?composite contains: ?part at: ?begin)
```

Fig. 5. Behavioural Visitor detection rule

method with selector `?visitSelector`. There must also be a class bound to the `?element` variable which must implement a method named `?accept` implemented as `?acceptBody`. The Visitor protocol demands that this method is called with the visiting object as its argument which is stated on the last line

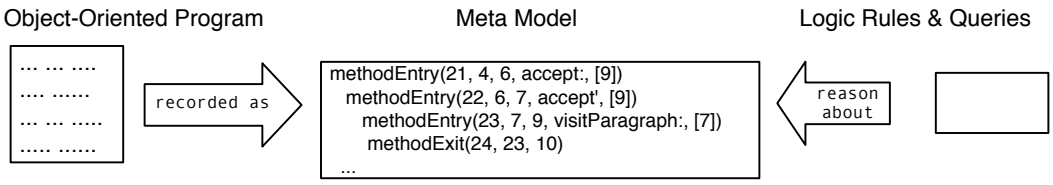

Fig. 6. An overview of Logic Meta Programming with a behavioural meta model

of the rule. In the body of the `?accept` method, a message `?visitSelector` must be sent back to the visiting object with the visited element as its argument. This is verified by matching the method's source code with the statements in the `methodStatements` part of the rule.

While the above rule is effectively used in the general approach to design pattern detection using LMP, the highly dynamic nature of the Visitor design pattern demands a different kind of detection based on the behavioural perspective.

First of all, the static nature of the current meta model creates a strong dependency on the actual implementation of the pattern in the source code with little room for small deviations. The above rule assumes for instance that the descend through the object structure is controlled by the visitor instead of by the object structure itself. This is evidenced by the body of the `?accept:` method which is required to match the following statement list exactly: a return of the result of the `?visitSelector` message being sent to the `?visitor` class. In order to be able to detect the implementation variant in which the control over the descend is located in the visiting object, an additional logic rule must be defined. This solution is far from elegant nor efficient.

Furthermore, the above deviation isn't the only one possible. The logic rule also demands that the literal symbol `self` is passed as an argument to the `?accept` method. Another implementation variant might pass the visited object indirectly through a method invocation and will thus remain undetected.

While logic meta programming with a static meta model can be used to detect a pattern based on the structural perspective, the independence from exact source code allows for a more expressive and flexible formulation of logic pattern detection rules in the behavioural perspective. In the following section we will explain how our current meta model can be modified to allow pattern detection from the behavioural perspective.

5 Detection of the Visitor Behaviour

As mentioned in the previous section, we would like to detect the Visitor design pattern by the behaviour it exhibits at run-time. The inclusion of behavioural information however demands a modification of the meta model according to which logic facts represent the program under investigation. Our ad-hoc

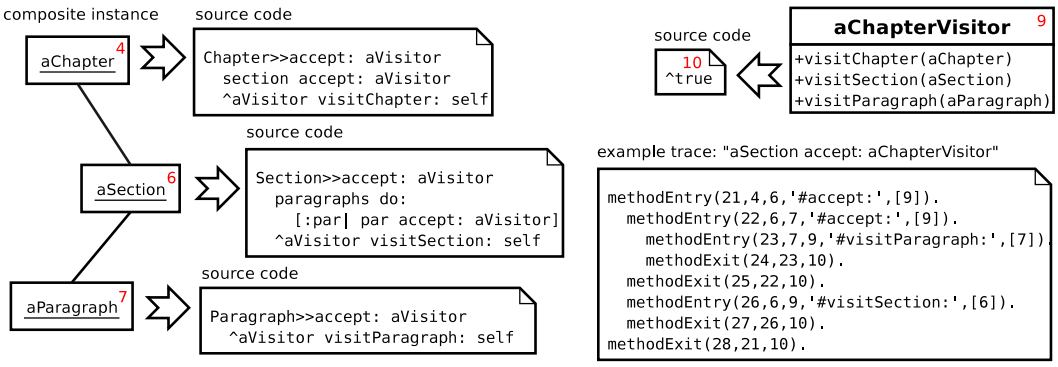

Fig. 7. Example of a composite object structure visited by a `ChapterVisitor` and extract of the corresponding execution history.

approach to a more dynamic meta model is based upon execution traces. In contrast to the situation depicted in figure 1, our new meta model models a program as an ordered collection of execution events that occur at run-time. Gathering these events requires the program to be executed. Figure 6 gives a general overview of this approach to logic meta programming.

We record three kinds of execution events: method invocations, variable assignments and method exits. An invocation is modelled by the `methodEntry` predicate which records the order of the event in the execution history, the instance sending the message, the receiving object, the method selector and the method's arguments:

```
methodEntry(?sequenceNumber, ?sendingInstance,
            ?receivingInstance, ?receivedSelector,
            ?receivedArguments)
```

The other execution events are modelled by similar logic predicates. Figure 7 depicts an example program with the corresponding execution trace.

Previously, we demonstrated how instances of the visitor design pattern can be detected using the structural meta model by searching for literal translations of the pattern's architecture in a program's source code. Using the new behavioural meta model, we can however also detect instances of the pattern using the succinct rule shown in figure 5 which is a straightforward translation of the corresponding sequence diagram shown in figure 3.

The `visits:from:till:invokedBy:` predicate expresses that an `?invoker` object caused a `?visitor` object to visit a `?composite` from the `?begin` sequence number corresponding with the `accept:` method invocation till the `?end` sequence number corresponding with the matching method exit event if two conditions are met. The first condition captures the double dispatching event of the pattern: there should be a double dispatching between the `?composite` and the `?visitor` in which the first method plays the role of the accept method in the pattern –the `?acceptselector` is received by a `?composite` with the `?visitor` as its argument– and the second method behaves as the visit method in the pattern – the `?visitselector` is received

by the `?visitor` with the `?component` as its argument. The second condition captures the recursive nature of a visitor: in addition to the presence of the above *double dispatching* pattern, we also demand that the visitor *recursively* visits all the components of the composite.

The results of the query

```
if ?visitor visits: ?composite from: ?begin till: ?end
```

on the execution trace shown in figure 7 are shown below:

?composite	?invoker	?begin	?end	?visitor
a Chapter	a VisitorInvoker	20	30	a ChapterVisitor
a Section	a Chapter	21	27	a ChapterVisitor
a Paragraph	a Section	22	24	a ChapterVisitor

From these results we derive that the `ChapterVisitor` class (the root of a class hierarchy of visitors for transforming formatted book chapters to for instance plain text files) visits the `Chapter` class beginning with method invocation 20 ending with a method exit at sequence number 30. During this visitation, the visitor also pays a visit to the `Section` class from sequence number 21 till sequence number 27. The `Paragraph` class is visited from sequence number 22 till 24.

The recursive nature of the Visitor design pattern is emphasised by the order in which the components are visited: the visitation of the root node ends when the invocation of the visitor on its children has ended. The control over the recursive descend of the composite structure is located in the structure itself which can be derived from the solutions by observing that the `?invoker` variable is always bound to the parent node in visitations originating from higher levels in the structure. This basic example of the Visitor design pattern can thus not be detected using the original structural rule.

This rule is by nature insensitive to differences in common implementation variants of the visitor as they mostly exhibit the same run-time behaviour. For instance, it is insignificant whether the visitation of composite elements happens through an iteration over elements in a collection (as is the case for the `Section` class) or through a (possibly indirect) call to an instance variable (as is the case for the `Chapter` class).

6 Towards Abstract Interpretation for Design Recovery

Together with experience gained from further experiments with a behavioural meta model [1], the Visitor design pattern example from the previous section indicates the need for a behaviour-based meta model for LMP. Other experiments performed included the detection of the accessor method pattern, which is more succinctly expressed using the behavioural meta model because of variations such as lazy initialisation of the accessed variable. However, as indicated proving the presence of these patterns in the behavioural meta

model is currently done using an extraction of the model based on tracing the execution of the program. This introduces the problem that this information is only valid for one of many different possible program executions. As our current behavioural meta model doesn't generalise over all possible execution paths, it doesn't allow the existence of a pattern to be proven with mathematical certainty. Our position is that abstract interpretation could provide a founding model for behavioural-based LMP for design recovery.

Abstract interpretation allows behavioural information to be derived mathematically from an approximation of the actual program semantics. As our current ad-hoc behavioural meta model has shown, this kind of information can greatly improve the effectiveness of pattern detection in logic meta programming. Therefore, our initial future work involves further investigating which behavioural program properties are needed for pattern detection and which can be derived through abstract interpretation. The granularity of the behavioural information that can be obtained through abstract interpretation will greatly determine the kind of software patterns that can be detected. Some patterns require for instance knowledge about specific object identities while for other patterns general information about the classes involved suffices.

We will also investigate whether the abstract interpretation process can be kept separate from the pattern detection process or whether they are to be interleaved. In the former case, our logic programs will just be reasoning about facts gathered by an abstract interpreter. In the latter case, the reasoning process might determine exactly what kind of program properties are needed to prove the existence of a pattern and configure an abstract interpreter tailored for that exact purpose.

We would also like to investigate whether abstract interpretation not only enables us to prove that a certain pattern is present in every possible program execution, but also whether a pattern is only present under certain conditions. An exact determination of the conditions under which a pattern is present would greatly surpass the current possibilities of logic meta programming.

Finally, approximate reasoning is another path we are pursuing in parallel in our research for a more flexible detection of software patterns [1]. Our initial experiments have shown that approximate reasoning aids in overcoming small discrepancies between the facts needed to prove the existence of a pattern and the program facts at hand. Later research might also entail investigating how these two approaches can be combined conceptually: approximate reasoning about approximate program semantics.

References

[1] Coen De Roover. Incorporating dynamic analysis and approximate reasoning in declarative meta-programming to support software re-engineering. Licentiate's thesis, Vrije Universiteit Brussel, 2004.

[2] Johan Fabry and Tom Mens. Language-independent detection of object-oriented design patterns. In *Elsevier International Journal: Computer Languages, Systems and Structures*, 2003.

[3] Peter Flach. *Simply Logical*. John Wiley & Sons, 1994.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable Object-Oriented software*. Addison-Wesley, 1995.

[5] Adele Goldberg and Dave Robson. *Smalltalk-80: the language*. Addison-Wesley, 1983.

[6] Kris Gybels. Soul and smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, 2003.

[7] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th SEKE Conference*, 2001.

[8] Tom Mens and Tom Tourwé. A declarative evolution framework for object-oriented design patterns. In *Proceedings of Int. Conf. on Software Maintenance*, 2001.

[9] Lutz Prechelt, Barbara Unger-Lamprecht, Michael Philippsen, and Walter F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Trans. Softw. Eng.*, 28(6):595–606, 2002.

[10] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA 1998*, 1998.

[11] Roel Wuyts. *A Logic Meta Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[12] Roel Wuyts and Kim Mens. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 1999*, 1999.