Generating Design Pattern Detectors from Pattern Specifications

Dirk Heuzeroth, Stefan Mandel
Universität Karlsruhe
Program Structures Group
76133 Karlsruhe, Germany
heuzer@ipd.info.uni-karlsruhe.de

Welf Löwe
Växjö universitet
MSI, Software Tech. Group
Växjö, Sweden
Welf.Lowe@msi.vxu.se

Abstract

We present our approach to support program understanding by a tool that generates static and dynamic analysis algorithms from design pattern specifications to detect design patterns in legacy code. We therefore specify the static and dynamic aspects of patterns as predicates, and represent legacy code by predicates that encode its attributed abstract syntax trees. Given these representations, the static analysis is performed on the legacy code representation as a query derived from the specification of the static pattern aspects. It provides us with pattern candidates in the legacy code. The dynamic specification represents state sequences expected when using a pattern. We monitor the execution of the candidates and check their conformance to this expectation.

We demonstrate our approach and evaluate our tool by detecting instances of the Observer, Composite and Decorator patterns in Java code using Prolog to define predicates and queries.

1. Introduction

Understanding a system's *design* and *architecture* [5] is essential for its maintenance, re-engineering, and further development. Since the software systems may be large and complex, and their design is almost always scarcely documented or even not available, software *comprehension* should be supported by tools. Design, however, is hard to detect automatically, since it is not tangible. Fortunately, standard solutions to solve certain design problems have been established. These *design patterns* are described together with the design problem(s) they intend to solve [4]. Many of them have been formalized, for example to generate corresponding templates [13]. If it were possible to detect these patterns in software systems, one would be able to deduce the intended design. This also helps to infer the system's architecture, since connectors are often implemented with design patterns – a fact we exploit already in our re-engineering and component adaptation approach [9].

Design patterns usually have static aspects (structural connections) and dynamic aspects (protocol of actions). Hence, we propose to analyze systems with respect to both the static and dynamic aspects combining static and dynamic analyses. The combination is necessary, because there are situations where neither static nor dynamic analyses alone are sufficient (or not with acceptable expenses), e. g. when user interactions occur. Our approach thus requires the programs' source code to be available, and the programs to be executable to observe their dynamic interaction aspects. We explicitly excluded all dependencies to coding and naming conventions.

In our previous papers [7, 6], we implemented static and dynamic analyses to detect instances of the Observer, Composite, Mediator, Visitor, and Chain of Responsibility design patterns. In [8], we also presented how to integrate visualizations.

An inconvenience concerning our previous work is, that we needed to implement a different detection algorithm for each design pattern, manually. Although the adaptation effort has been very limited due to the similar structure and protocols of many patterns, we want to improve the situation by generating the analysis algorithms from specifications of design patterns.

2. Approach

To detect design patterns in source code, we first perform a static analysis that provides a set of candidate pattern instances conforming to the static structure of the patterns to be detected. A pattern candidate is a tuple of program elements each in a certain role with respect to the pattern. An example of such a tuple for an instance of an Observer pattern is:

```
(ConcreteSubject.addObserver,
 ConcreteSubject.detachObserver,
 ConcreteListener.notifyObservers,
 ConcreteListener.updateObserver)
```

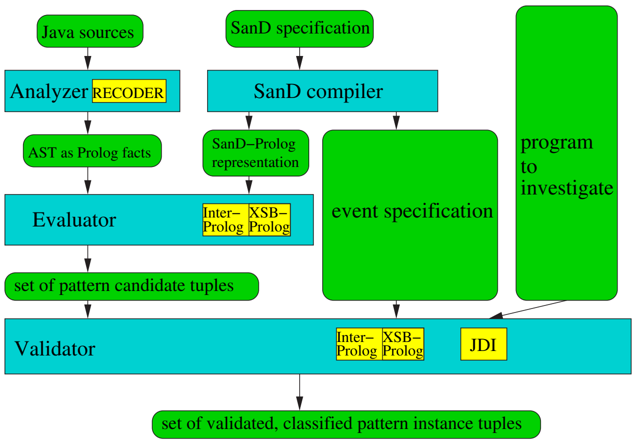

Figure 1. Pattern detection framework.

We distinguish the *role of a program element* in a pattern from the program element itself. `Subject` and `Listener`, e. g., are roles of classes in the Observer pattern. `ConcreteSubject` and `ConcreteListener` can be two concrete classes in the two roles. Similarly, a concrete method `addObserver` could play the `attach` in an Observer pattern.

We also have to distinguish an *object instance* from a *design pattern instance*, if the latter contains class program elements. For example we could have different object instances of `ConcreteSubject` each with several object instances of `ConcreteListener`, or, even `AnotherConcreteListener` attached to it.

The dynamic analysis monitors the execution of the legacy program. It classifies the candidate tuples proposed by the static analysis according to their conformance to the expected pattern behavior. The categories are *full match* (full protocol conformance), *may match* (a prefix of a correct protocol could be detected, the candidate does not violate the protocol though), *mismatch* (violation of the protocol) and *no decision* (candidate not executed).

We provide two languages to specify design patterns allowing to generate the static and dynamic analyses automatically: a powerful, extensible, low-level language called SanD-Prolog[1] (Section 2.1) consisting of Prolog predicates that specify the static structure and dynamic behavior of design patterns and a high-level language SanD[2] (Section 2.2) that allows to specify design patterns even more abstract and intuitively. We implemented a simple compiler that transforms SanD specifications into SanD-Prolog specifications. In Figure 1 shows the structure and data flow of our pattern detection system.

[1]Static and Dynamic Prolog Specification
[2]Static and Dynamic Specification Language

2.1. The Specification Language SanD-Prolog

Basically, SanD-Prolog is a collection of Prolog predicates, that represent relations between source code elements (attributed AST nodes) as well as the states and state transitions of the protocol defined by a pattern.

The static specification contains Prolog predicates to identify the types of syntax elements like classes, methods, calls, etc. and relate them to the roles in the pattern. Below you see (a part of) the static specification for the Observer pattern:

```
observer(Vattach,Vattachee,Vdetach,Vdetachee,Vlistener,
         Vnotify,Vsubject,Vupdate) :-
  listener(Vlistener,Vupdate),
  subject(Vattach,Vattachee,Vdetach,Vdetachee,Vlistener,
          Vnotify,Vsubject,Vupdate).

subject(Vattach,Vattachee,Vdetach,Vdetachee,Vlistener,
        Vnotify,Vsubject,Vupdate) :-
  notify(Vnotify,Vsubject,Vupdate),
  attach(Vattach,Vattachee,Vlistener,Vsubject),
  detach(Vdetach,Vdetachee,Vlistener,Vsubject),
  class(Vsubject).

attach(Vattach,Vattachee,Vlistener,Vsubject) :-
  attachee(Vattachee,Vlistener),
  assignAttachee(Vattachee,Vstatement15),
  member(Vattach,Vsubject),
  method(Vattach),
  parameter(Vattachee,Vattach),
  statement(Vstatement15,Vattach).

...
```

We also represent the dynamic behavior of patterns by predicates based on the temporal logic of actions (TLA) [11]. Here the predicates specify the relevant states and state transitions of a pattern's behavior to detect in a concrete program run. This specification corresponds to a simulation of the pattern that our dynamic analysis performs. This analysis checks if a concrete program run of the pattern candidate conforms to the specified pattern behavior.

The dynamic analysis is based on events triggered during candidates executions. The event triggering points are specified by `watch` predicates used to set breakpoints in the monitored program execution. Dynamic constraints checked at certain breakpoints are defined by a corresponding Prolog procedure. The code below shows an example for a watch predicate specifying a breakpoint for the execution of methods in the `attach` role, and a procedure `onMethodEntry` checking the corresponding dynamic constraints:

```
watch('attach',Vattach,Arguments) :-
  observer(Vattach,Vattachee,_,_,_,_,_,_),
  Arguments = ['this',Vattachee].

onMethodEntry('attach',Vattach,[Vsubject,Vattachee]) :-
  dynamicConformTyped(Vattachee,VattacheeClass),
  containingTyped(VattacheeName,VattacheeClass),
  dynamicConformTyped(Vsubject,VsubjectClass),
  dynamicObserver(VNo,Vattach,VattacheeName,_,_,_,_,
                  VsubjectClass,_),
  request(assert(attached(VNo,Vsubject,Vattachee))),
  fail.
```

The predicate `onMethodEntry` specifies the expected dynamic behavior of an execution of the specified attach method as a conjunction of calls to further Prolog procedures. The arguments to the parameters of the procedure `onMethodEntry` are supplied during dynamic analysis by querying the runtime information at the breakpoint when execution enters the method in attach role. Each of the supplied arguments has a unique dynamic identifier. The procedure `dynamicConformTyped` transforms these identifiers into their static equivalent, by specifying that the second parameter has to be the same class or a superclass of the first parameter. The second procedure call `containingTyped(...)` associates the static variable identifier `VattacheeName` to the previously retrieved class identifier `VattacheeClass`. `VattacheeName` then contains the static identifier for the parameter in the attachee role. The next step retrieves the dynamic pattern candidate tuple that corresponds to the object instance of the pattern identified by the elements calculated so far. This is done by calling the `dynamicObserver` with the appropriate arguments. Matching tuples are identified by their number `VNo`. For each match the procedure

```
request(assert(attached(VNo,Vsubject,Vattachee)))
```

is called. It sets the relation

```
attached(VNo,Vsubject,Vattachee)
```

to true, and thus simulates attaching `Vattachee` to `Vsubject`. An object instance of the pattern is thus represented by the predicates that simulate its states and execution, together with the concrete values of the participating variables that represent concrete objects. The Prolog interpreter considers each match and every possible combination of the values specified by the constraints represented by the previously given procedure calls due to the final call to the `fail` procedure.

So, SanD-Prolog is suited to specify the static and dynamic constraints of design pattern, but specifications might become complicated and lengthy so that the pattern is hard to recognize in the crowd of predicates. We therefore introduce our more intuitive specification language SanD, now.

2.2. The Specification Language SanD

The SanD specification language integrates the specification of the dynamic behavior of a pattern into the specification of its static structure. The notation for the specification of the static structure is similar to object oriented languages familiar to developers. The code below shows an example specification for the Observer:

```
pattern observer {

  class subject {
    method attach (attachee:listener-)
    [:attached(<this>,attachee);] {
      assign attachee | call passing attachee
    }
    optional method detach (detachee:listener-)
    [:~attached(<this>,detachee);] {
      assign detachee | call passing detachee
    }
    method notify
      [:~updated(<this>,*);] {
        call update
      }
      [updated(<this>,*) & attached(<this>,L) &
       ~updated(<this>,L) : discard;]
  }

  classOrInterface listener {
    method update { }
    [:updated(<stack>,<this>);]
  }
}
```

The dynamic specification consists of constraints defined by bracketed lists of `guard:assert` constructs attached to elements of the static specification. The `guard` expression specifies a condition that must hold in the current state of the program simulation, and the `assert` expression specifies a new state of the program simulation. The dynamic specification constructs, together with their association to a static constraint, correspond to dynamic specifications of SanD-Prolog.

In our example specification, the dynamic specification `[:attached(<this>,attachee);]` related to the method in `attach` role specifies that in the new state of the simulated program the parameter object `attachee` is attached to the subject object identified by `<this>`.

The dynamic specification attached to the method exit of the specified method in the `notify` role (see code above) states that the protocol is violated and the currently inspected candidate tuple has to be discarded if at least one listener has been updated, but there also exists an attached listener that has not been updated.

Obviously, pattern specifications are preferably written in SanD as this notation is more concise and intuitive than SanD-Prolog. However, SanD is less powerful than SanD-Prolog. For example, we cannot express constraints like class X must *not* contain method m. Nevertheless, we have been able to specify standard design patterns like Observer, Composite and Decorator patterns in SanD without problems. This might indicate that SanD could be sufficient in practice. The validation of this assumption, however, needs more expertise with the new language and will be matter of further research.

3. Evaluation

The static analysis is performed by the evaluator module on the Prolog database representing the source programs' attributed abstract syntax trees by a Prolog query corresponding to the predicates of the SanD-Prolog specification representing static pattern aspects. The dynamic analysis monitors the execution of the program to analyze and validates the behavioral specification by simulating the execution of a pattern instance on the specified states using the specified state transitions both associated to the executed program element. Violation of the constraints leads to rejection of the candidate instance.

We evaluated our approach by detecting Observer, Composite and Decorator patterns in (the Java code of) our own analysis tool, c. f. Table 1.

Table 1. Pattern Detection by SanD

pattern	static match	full match	partial match	no decision	correct
Observer	18	2	6	4	2
Composite	4	1	0	0	1
Decorator	2	2	0	0	2

Table 2 compares the execution times of the static and dynamic analyses of SanD and our previous hand-coded pattern detector MetaD [7] applied to the same program. The static analysis of SanD is much faster than that of MetaD, especially when there are several patterns to detect. SanD can do this simultaneously whereas MetaD has to be started once per pattern. On the other hand the dynamic analysis of MetaD is much faster than that of SanD, mainly because MetaD uses code instrumentation instead of the Java Debug Interface (JDI) to obtain runtime information. However, this is not inherent to our approach. The performance of SanD can be improved by using instrumentation instead of the JDI.

Table 2. Performance comparison

pattern matcher	SanD	MetaD
static analysis (Observer)	2 sec	4 sec
dynamic analysis (Observer)	20 sec	1 sec
static analysis (3 patterns)	2 sec	12 sec
dynamic analysis (3 patterns)	26 sec	3 sec

4. Related Work

Other approaches to detect patterns [14, 3, 10] mostly restrict themselves to static analyses using rather strong static signatures. These approaches fail to detect behavioral patterns as their static patterns are not distinctive enough. Brown [1] and Carriere etal. [2] perform a dynamic analysis, but either only at periodic events, or merely to identify communication primitives.

Alternative specification languages have been investigated before developing SanD: UML including OCL [**?**] does not provide the means we need to generate the analysis algorithms. LePUS [**?**] does not offer means to specify dynamic behavior. DisCo [13] is formal and treats static as well as dynamic aspects, but does not offer a mechanism to reject incorrect pattern candidates.

5. Conclusions and Future Work

We presented how to generate static and dynamic analyses to detect design patterns in legacy code starting from design pattern specifications. We proposed two languages both suited to generate the analyses: an intuitive comprehensible high-level design pattern specification language SanD and a more powerful but less intuitive low-level specification language SanD-Prolog. We validated our approach and tool by several case studies.

Future work comprises to study the expressive power of our languages in further case studies. We will improve the performance of the dynamic analyses by code instrumentation, and improve coverage by testing technology.

References

[1] K. Brown. Design reverse-engineering and automated design pattern detection in smalltalk, 1997.

[2] S. J. Carriere, S. G. Woods, and R. Kazman. Software Architectural Transformation. In *WCRE 99*, October 1999.

[3] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *TOOLS 30*, pages 18–32. IEEE Computer Society, 1999.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[5] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing, 1993.

[6] D. Heuzeroth, G. Högström, T. Holl, and W. Löwe. Automatic Design Pattern Detection. In *11th IWPC*, May 2003.

[7] D. Heuzeroth, T. Holl, and W. Löwe. Combining Static and Dynamic Analyses to Detect Interaction Patterns. In *IDPT*, Jun 2002.

[8] D. Heuzeroth and W. Löwe. *Software-Visualization - From Theory to Practice*, chapter Understanding Architecture Through Structure and Behavior Visualization. Kluwer, 2003.

[9] D. Heuzeroth, W. Löwe, A. Ludwig, and U. Aßmann. Aspect-Oriented Configuration and Adaptation of Component Communication. In *3rd GCSE*. Springer, 2001.

[10] R. K. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-based reverse-engineering of design components. In *ICSE*, pages 226–235, 1999.

[11] L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[12] A. Ludwig, R. Neumann, U. Amann, and D. Heuzeroth. Recoder homepage. `http://recoder.sf.net`, 2001.

[13] T. Mikkonen. Formalizing Design Patterns. In B. Werner, editor, *20th ICSE*, pages 115–124. IEEE, Apr. 1998.

[14] L. Prechelt and C. Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *J.UCS*, 4(12):866ff, 1998.