Design pattern detection is a reverse engineering methodology that helps software engineers to analyze and understand legacy software by recovering design decisions and thereby providing deeper insight into software. Recent research has shown that a combination of static and dynamic source code analysis can produce better results than purely static approaches. In this paper we present an extension of the pattern detection approach proposed by Wendehals [22]. In particular, we extend the specification language for behavioral patterns to increase its expressiveness and the approach’s recall by introducing the concept of set objects.

1. INTRODUCTION

Due to requests for new features and the discovery of defects, software has to be continuously adapted and maintained during its life cycle. Incomplete documentation or the unavailability of the original developers often complicate this task and are among the reasons that software engineers often spend more time to maintain a complex software system than to actually develop it. According to Sommerville 50% to 75% of the total programming effort spent on a system are devoted to maintenance [17].

The tedious and error-prone task of understanding a large system can be supported and simplified by reverse engineering tools that recover the design of the software and try to locate the application of design patterns. Identifying these pattern instances can help the reverse engineer to quickly understand a software system and thereby speed up the maintenance process. Design patterns were first introduced by Gamma et al. and are among the reasons that software engineers often spend more time to maintain a complex software system than to actually develop it. According to Sommerville 50% to 75% of the total programming effort spent on a system are devoted to maintenance [17].

The remainder of this paper is organized as follows: First, we give a general overview of the pattern detection process by Wendehals. In Section 2 we present an example and use it to demonstrate the shortcomings of the pattern specification language. A suitable extension of the language is proposed in Section 3. Section 4 deals with the realization of the approach that is then evaluated on a real software system in Section 5. In the subsequent Sections we discuss related work, draw conclusions and sketch ideas for future work.

2. STATIC AND DYNAMIC DESIGN PATTERN DETECTION

There are many examples in literature that use only a static source code analysis to recover design patterns (e.g. [1, 9, 15, 16, 18]). Although a static analysis is not limited to considering structural properties of the software under analysis certain object-oriented concepts like polymorphism and dynamic method binding make a precise static behavior analysis impossible. Hence, a common drawback of these approaches is that they generate false positives when design patterns have a similar structure. The State and Strategy patterns [6] are good examples for this: Their static structure is identical and they differ only in their runtime behavior. Common static pattern detection approaches often recognize implementations of the State pattern also as Strategy pattern implementations and vice versa. Obviously one of those results is always a false positive.
Figure 1 shows the pattern detection process as proposed by Wendehals [22]. It uses the source code of the software system and a library of structural patterns to carry out a static analysis. In his approach, that builds on the work in [10] and [11], graph grammar rules are used for the specification of structural patterns (cf. Section 4). The result of the static analysis is a set of possible implementations of design patterns, the so-called pattern candidates. Pattern candidates are sections in the source code whose structure corresponds to the structural patterns used in the analysis. Due to structurally similar patterns, the result set may contain many false positives. At this point a dynamic analysis is used to confirm or reject the candidates.

After detecting the pattern candidates, the software system under analysis is executed manually and the candidates’ behavior is traced. Depending on how often a candidate’s classes are instantiated during execution time, a number of traces is generated for each candidate. The candidates’ expected behavior is described with behavioral patterns based on UML 2.0 sequence diagrams [13]. During dynamic analysis the traces are compared with the corresponding behavioral patterns. If the majority of a candidate’s traces match the behavioral pattern, it is likely that the candidate is an actual design pattern implementation and the candidate is confirmed. If most of the traces for a candidate do not match the behavioral pattern, it probably is a false positive and thus rejected.

2.1 Example

Throughout this paper we use the Observer pattern to explain the pattern detection approach devised by Wendehals and our extension. Gamma et al. describe the Observer pattern’s intent as follows:

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." [6]

Figure 2 shows the structural description of the Observer pattern in the notation introduced in [11] and [12]. There are two classes subjectClass and observerClass. The subject class has the methods register, which takes an observer object as parameter, and notify. The observer class has an update method. The ellipses are so-called Annotations and refer to subpatterns which are specified in other diagrams. Annotations represent instances of required subpatterns. In this case the annotations indicate that the subject class’ notify method must implement a delegation to the observer class’ update method. The methods update and register should be overridden. (The shown observer and subject classes are intended to be subclassed by concrete observers and subjects that implement their own specific behavior.) The dashed lines of the OverridingMethod annotations indicate that these subpatterns are not mandatory for the detection of the Observer pattern. The MultiReference annotation expresses that a subject references arbitrarily many observers. Finally, the Observer annotation that is marked with create is created when the depicted structure is found in the software system under analysis. It tags the structure as candidate for the Observer pattern.

Figure 3 illustrates the expected behavior of the Observer pattern in the syntax defined in [22]. It shows one object a of the type subjectClass and two observer objects a and b of the type observerClass. The types refer to object names from the structural pattern (cf. Figure 2). a and b both call the register method of the subject class to register themselves for the subject’s updates. The following loop fragment indicates that the enclosed message sequence must occur at least once but can occur arbitrarily many times. It states that whenever the subject class calls its notify method each observer’s update method has to be called. If an observer candidate fails to show this behavior it probably is a false positive (or a variation of the design proposed in [6]).

For details on the specification of behavioral patterns with sequence diagrams we refer to [20]. More on the dynamic pattern detection algorithm can be found in [23].

2.2 Shortcomings of the Approach

One problem of the described approach lies in the use of absolute quantities of objects where in reality arbitrarily large sets of objects can participate in the pattern. The behavioral pattern in Figure 3 uses an exemplary situation with two observer objects to specify the desired behavior of an implementation of the Observer pattern. In reality any number of observer objects could communicate with the subject as long as the messages occur in the correct order. The behavioral analysis algorithm is currently limited to only recog-
nize traces with exactly the specified number of objects as correct. In the example only candidate instances with two observer objects would be deemed accurate. Traces that represent the same situation with any other number of objects and otherwise conform to the behavioral pattern are rejected. This increases the probability that a candidate is incorrectly labeled as a false positive.

The same problem arises for all other patterns that involve a possibly arbitrarily large set of objects. Examples are the State pattern (an object can be in one of arbitrarily many states) and the Chain of Responsibility pattern (a request is passed down an arbitrarily long chain of handler objects until one handler consumes it) [6].

3. EXTENDING BEHAVIORAL PATTERNS WITH SET OBJECTS

Our approach solves the mentioned shortcomings by introducing a new element to the behavioral pattern specification language: the Set Object. A set object represents an arbitrarily large set of objects of the same type. With this new construct the Observer pattern can be specified without the need to predefine the exact number of observer objects involved at execution time.

Figure 4: Observer behavioral pattern with set object

Figure 4 shows the Observer behavioral pattern with the new element. The set object is depicted by the double border and replaces the two separate observerClass objects from Figure 3.

The introduction of the set object necessitates proper semantics for messages between set objects and regular objects. A message from a set object to a regular object represents a method call from one of the objects in the set to the regular object. An example for this is the register message in Figure 4. Analogously, a message in the opposite direction represents a call to one object that is of the same type as the set object.

Additionally, we need to model the case that a method is called on each object in a set. For this we introduced a new combined fragment, the Each Fragment, which has semantics similar to the loop fragment. An each fragment can only be used for messages from regular objects to set objects (or vice versa) and means that the contained messages are sent to each object in the set (or from each object in the set to the regular object). Note that the semantics for message passing from one set object to another in conjunction with an each fragment remains undefined here. If such a construct was allowed, it would have to be specified if a message should be passed from each object in set 1 to each object in set 2 or if other combinations would be appropriate. However, in our investigations we have not found a case where such a construct would be needed. The exploration of this topic remains future work.

In Figure 4 the each fragment is used to express that after the subject has called its own notify method, it must call the update method of each of its observers.

4. REALIZATION

The Reclipse tool suite [21] has been implemented as a collection of plug-ins for Fujaba4Eclipse, which is an integration of Fujaba [4] into the eclipse framework.

Structural patterns are specified as special graph grammar rules. They describe the object structure that constitutes a given pattern in abstract syntax. These graph grammar rules are then translated to Story Diagrams [3] from which code is generated. The generated code realizes search algorithms for every structural pattern. The software under analysis is parsed into an abstract syntax graph representation. The search algorithms try to match the patterns in the abstract syntax tree and create annotations when a matching object structure is found (cf. Section 2). The annotations mark objects that represent relevant roles of a given pattern, e.g. for the Observer pattern the subject and the observer objects. Details on the structural pattern detection process can be found in [10].

In order to analyze the runtime behavior of a software system it is executed and the method calls that occur during execution are traced. To reduce the amount of data that has to be analyzed, not the complete behavior of the software is traced but only the instance behavior of previously annotated classes and methods, i.e. of pattern candidates that were identified during structural analysis. The behavior of objects of other types and other method calls is omitted. All pattern candidates are traced individually and, depending on the concrete program execution, a number of traces is generated for each of them.

The behavioral analysis algorithm assesses if each candidate’s traces conform to the corresponding behavioral pattern. A trace conforms to the behavioral pattern when its method calls all conform to the pattern and when the trace represents a complete pass through the pattern. In this case the trace is accepted. If the trace contains method calls that violate the behavioral pattern, it is rejected. If the trace does not contain prohibited method calls but does not contain all mandatory method calls that are specified in the pattern, the software system may not have been executed long enough to collect sufficient data. In this case the trace is neither rejected nor accepted. Technically the analysis is performed by mapping the behavioral patterns to finite automata and using the traces as input for the automata. The traces can be accepted, not accepted or rejected by an
the static model while the Chain of Responsibility makes pattern specifications easier at the expense of their expressiveness. Conditions like "a class may not have any methods" cannot be expressed in SAND-PROLOG.

7. CONCLUSIONS AND FUTURE WORK
In this paper we presented an extension of the pattern detection approach described by Wendehals [22]. We introduced a new element to the pattern specification language to be able to deal with arbitrarily large sets of objects in behavioral pattern specifications. The extension was implemented for the RECLiPSe tool suite and evaluated for several patterns. It is now possible to correctly detect patterns that caused problems in Wendehals' original approach [14].

The approach still leaves open questions for future research. We intend to analyze larger software projects and try to detect more patterns to get a feeling for the scalability and expressiveness of our approach. Furthermore it would be interesting to quantitatively analyze the precision and recall of the extended RECLiPSe tool suite and compare it to similar reverse engineering tools. In the future we want to build upon the current reverse engineering techniques and use reverse engineered behavioral models to carry out further analyses like conformance checking.

8. REFERENCES
[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, USA, 1995.


APPENDIX

A. BEHAVIORAL PATTERN

CHAIN OF RESPONSIBILITY

The Chain of Responsibility design pattern is another pattern where set objects can be used. The pattern intent is described in [6] as follows:

"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it." [6]

![bp Chain of Responsibility](image)

Figure 5: Chain Of Responsibility behavioral pattern

In Figure 5 there are two objects. `client` is an untyped object that delegates the request `r` to the handler chain. A set object represents an arbitrary number of `AbstractHandler` objects that constitute the chain. The `handleRequest` message represents the passing of the request along the chain.

Note the property `{other}` on the message, which is another language extension presented in [14]. A message from a set object to itself can either represent a method that is called by an object in the set on itself (i.e. the caller instance equals the callee instance) or a call of that method on another object in the set. To distinguish between these cases we introduced the keywords `self` and `other`.

The `other` call is enclosed by a loop fragment with the bounds 0 and `. Either the request is handled (and consumed) by the first handler in which case the loop fragment would be executed zero times or it is handled by an arbitrary handler somewhere in the chain.