# Recognizing Behavioral Patterns at Runtime using Finite Automata

Lothar Wendehals
Software Engineering Group
Department of Computer Science
University of Paderborn, Germany
lowende@uni-paderborn.de

Alessandro Orso
SPARC Group
College of Computing
Georgia Institute of Technology, USA
orso@cc.gatech.edu

## ABSTRACT

During reverse engineering, developers often need to understand the undocumented design of a software. In particular, recognizing design patterns in the software can provide reverse engineers with considerable insight on the software structure and its internal characteristics. Researchers have therefore proposed techniques based on static analysis to automatically recover design patterns in a program. Unfortunately, most design patterns comprise not only structural, but also significant behavioral aspects. Although static analysis is well suited for the recognition of structural aspects, it is typically limited and imprecise in analyzing behavior. To address this limitation, we present a new technique that complements our existing static analysis with a dynamic analysis, so as to perform a more accurate design-pattern recognition. The dynamic analysis is based on (1) transforming behavioral aspects of design patterns into finite automata, (2) identifying and instrumenting relevant method calls, and (3) monitoring relevant calls at runtime and matching them against the automata. The results of the dynamic analysis are then used to compute the likelihood of a pattern to be in the code. This paper describes our technique and presents a preliminary empirical study performed to assess the technique.

**Categories and Subject Descriptors:** D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement-*Restructuring, reverse engineering, and reengineering*;

**General Terms:** Algorithms

**Keywords:** Design pattern recognition, static and dynamic analysis, finite automata, tracing.

## 1. INTRODUCTION

Software engineers spend most of their time maintaining software systems. Software is modified to fix faults or meet new requirements. Because documentation is often not available or obsolete, engineers must reverse engineer a software before changing it to understand its internals and recover its design. In particular, identifying design pattern instances [6] can provide reverse engineers with considerable insight on the software structure and its internal characteristics. Design patterns are good design solutions to recurring problems and form a common vocabulary among developers. The recognition and documentation of design pattern instances aids program understanding by making the intention of a design explicit and, most importantly, supports effective reuse.

Manually identifying design patterns in the code is an extremely time consuming task. Therefore, researchers have proposed techniques based on static analysis to automatically recover design patterns in a program (e.g., [1, 8]) based on the patterns' structure. However, most design patterns comprise not only structural, but also significant behavioral aspects that must be considered when recognizing patterns. An implementation that satisfies the structural requirements of a pattern, while violating its behavioral requirements, can hardly be considered an instance of that pattern.

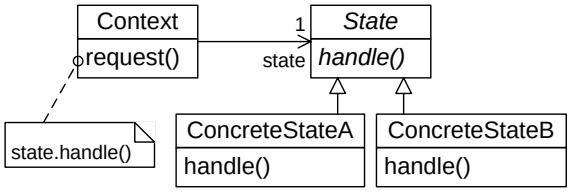


**Figure 1: The *State* Design Pattern**

The *State* design pattern (Figure 1), for instance, allows an object to alter its behavior depending on its internal state. An interface for handling the behavior is defined by the abstract class State and implemented by different ConcreteStates. A Context object delegates requests to the referenced State that handles them. Although the structure of this pattern is easily recognizable statically, there are behavioral constraints imposed by the pattern. For example, the state can be initially set by a client, but can be set only by the Context or ConcreteStates afterwards. To be an instance of the *State* pattern, an implementation should both match the structure and satisfy such behavioral requirements.

Moreover, different design patterns may have the same (or at least similar) structure. The *Strategy* design pattern (Figure 2), for example, has exactly the same structure as the *State* pattern, but a very different purpose. Without looking at behavioral constraints, it would be impossible to distinguish instances of these two patterns.

Static analysis can precisely identify the structural aspects of a design pattern, but even the most accurate analysis is typically too imprecise to be able to infer from the code the dynamic information needed to unambiguously identify


*WODA'06,* May 23, 2006, Shanghai, China.

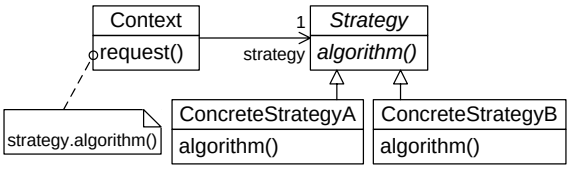


**Figure 2: The *Strategy* Design Pattern**

a design pattern. In most cases, this information can be computed only by observing, at runtime, the actual behavior of the instances of the classes participating in the pattern.

To overcome the limitations of a solely static analysis, one of the authors has proposed to combine static and dynamic analysis to improve the precision of the pattern-recognition process [14]. The previously defined static analysis [10] identifies a set of candidate design-pattern instances. The dynamic analysis checks at runtime whether the instances actually satisfy the patterns' behavioral requirements. More precisely, our dynamic analysis (1) transforms behavioral requirements of design patterns into finite automata, (2) uses the candidates computed in the static analysis to identify and instrument relevant method calls, and (3) monitors relevant calls at runtime, while the software is being used, and matches them against the automata. The results of the dynamic analysis are used to compute the likelihood of a pattern instance to be in the code. The engineer is then presented with a summary of both the static and dynamic matching results.

In this paper, we present our dynamic analysis. We also present a preliminary empirical study performed using the ECLIPSE framework [4] as a subject. We successfully identified several pattern instances in ECLIPSE, which could not have been identified unambiguously by a purely-static approach. Their presence is actually documented [5], which confirms that our results are accurate. Therefore, although preliminary, the results are promising in showing the potential effectiveness and applicability of our approach.

The paper is organized as follows. Section 2 provides an overview of our approach. In Section 3, the specification of behavioral requirements is explained. Section 4 describes how these behavioral requirements are transformed into finite automata, whereas Section 5 explains how method call traces are checked against the automata. Section 6 presents our empirical evaluation. Related work follows in Section 7. Finally, Section 8 summarizes ongoing and future work.

## 2. OVERVIEW OF THE APPROACH

Before describing our approach, we introduce the *pattern catalog*. The pattern catalog is a collection of pattern specifications. It contains specifications for many of the design patterns presented in [6], such as *State*, *Decorator*, and *Bridge*. For each pattern considered, the catalog contains two models: the structural pattern and the behavioral pattern. The former encodes the structural requirements for the pattern using a notation similar to UML object diagrams and expressed at the abstract-syntax graph level (see [10]). The latter encodes the behavioral requirements using UML sequence diagrams. The pattern catalog is the basis of our design recovery process that consists of three phases: pattern customization, static analysis, and dynamic analysis.

*Pattern customization.* Because the description of the patterns provided in [6] is not formal, developers can tailor the patterns to better suit their needs. To account for the existence of several implementation variants of a pattern, our approach provides reverse engineers with the possibility of customizing the structural and behavioral patterns to reflect specific variants. The customization is performed in an iterative fashion. First, the reverse engineer applies our static analysis using the default catalog to a part of the software. Then, based on the results, the engineer suitably adapts some of the patterns and repeats the first step. This process continues until the engineer is satisfied with the customized catalog. At this point, the next phase takes place and the analysis is applied to the whole software. Prior to the analyses, the structural and behavioral patterns are compiled into a format amenable to the analyses' algorithms.

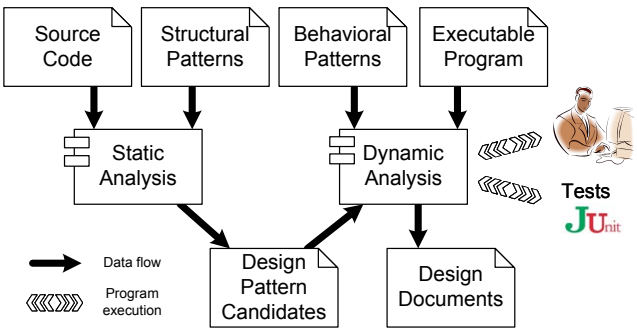


**Figure 3: The Design Recovery Process**

*Static analysis.* The static analysis (cf. Figure 3) inputs the source code of the system under analysis and the structural patterns. The source code is parsed into an abstract syntax graph that is searched for the structural patterns. Each match of a structural pattern is added to a set of pattern-instance candidates. Because the static analysis was defined in previous work [10], we do not further discuss it.

*Dynamic analysis.* The dynamic analysis inputs the executable program, the behavioral patterns, and the set of candidates. The goal of this phase is to check, for each candidate, whether the interactions of its constituents at runtime conform to the behavioral requirements of the pattern. The technique monitors method calls at runtime, while the program is executed either by the user or against a test suite.

Each method call trace that conforms to a behavioral pattern increases the confidence in the existence of the design pattern instance. Conversely, traces that violate a behavioral pattern decrease such confidence. The results for both kinds of traces are presented to the reverse engineer in the form of design documents (e.g., annotated class diagrams).

## 3. BEHAVIOR SPECIFICATION

As discussed above, we specify a structural and a behavioral pattern for each design pattern. A behavioral pattern encodes the dynamic aspects of a pattern using a sequence diagram. Our behavioral patterns take full advantage of the syntax elements introduced in UML 2.0 [15]. In particular, we use *combined fragments*, visualized as rectangles with an operator in the upper left corner, with various kinds of operators. For example, we use `opt` to define optional sequences, `loop (1,n)` to define a sequence executed at least once and up to `n` times, and `alt` to define alternative sequences.

To derive the initial definition of a behavioral pattern, we start from the (typically informal) description of the corresponding design pattern. For example, this is an excerpt

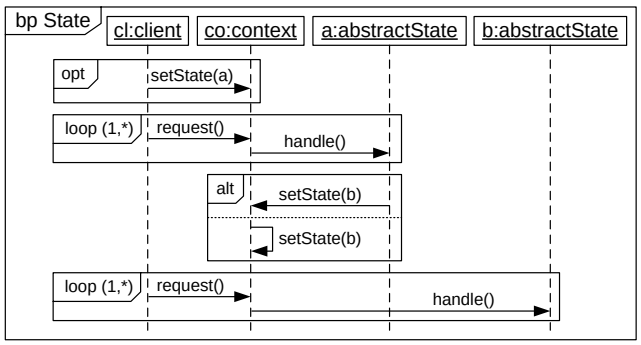


**Figure 4: The *State* Behavioral Pattern**

from the description of the *State* pattern provided in [6]: *"... Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly. Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances."*. From this description, we can infer the requirement that a state may be initially set by a client, but can only be changed by Context or ConcreteState objects afterwards. Another, somehow implicit, requirement is that the state should actually change during execution, for the notion of states to make sense.

Figure 4 shows the *State* behavioral pattern that encodes the above requirements. The behavioral pattern involves four objects, namely cl:client, co:context, a:abstractState, and b:abstractState. The opt operator indicates that cl:client can optionally configure the context initially by calling setState with a as argument. The cl:client object calls request on co:context, which in turn calls handle on a:abstractState. As specified by operator loop(1,*), this sequence must occur at least once, but may occur an arbitrary number of times.

Then, either the a:abstractState or the co:context itself must change the state by calling setState with b as argument. After the state change, the cl:client must call request on co:context, and co:context must call handle on b:abstractState. Also in this case, this sequence must occur at least once.

All names in the behavioral patterns are variables. The variables for classes and methods, such as context or request, are bound to concrete classes and methods during the static analysis. Conversely, object names, such as co and cl are bound to actual objects during the dynamic analysis.

Behavioral patterns do not require to consider all method calls that might occur at runtime. Only relevant method calls are specified. The matching between an execution and the pattern is performed by considering only calls of methods that are used within the pattern. Other methods calls are simply ignored and, thus, can occur in any order and interleaved with the method calls of interest. However, method calls that appear in the behavioral pattern must occur exactly in the order specified. This requirement complies with the semantics of the *consider* operator in UML 2.0, which implicitly holds for all behavioral patterns.

# 4. FROM BEHAVIORAL PATTERNS TO FINITE AUTOMATA

Behavioral patterns can be interpreted as regular expressions. The symbols used in these regular expressions consist of a combination of the method call, its caller, and callee. We use finite automata to match the regular expressions. In this section, we describe how behavioral patterns are transformed into Deterministic Finite Automata (DFA) that get method call events as input and check method traces for their conformance to the behavioral patterns.

To generate DFAs, we define, for each syntactic element of a behavioral pattern (e.g., method call, optional, and alternative fragment), a transformation into parts of a Non-Deterministic Finite Automaton (NFA). The complete NFA for a behavioral pattern is constructed by combining the NFA parts for each element of the pattern. The NFA is then transformed into a DFA. For the sake of space, we do not present the complete set of transformations for all syntactical elements. Instead, we illustrate a few of these transformations (Figure 5) to give the reader the intuition of how our transformation works. Specifically, we show how we (1) transform a method call that occurs in a behavioral pattern into its corresponding part of an NFA, (2) combine two partial NFAs that correspond to two consecutive calls, and (3) transform a loop into its corresponding NFA fragment.

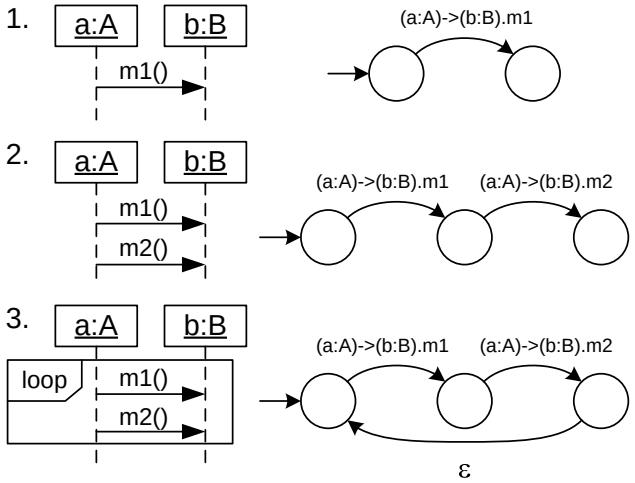


**Figure 5: Transformation of Behavioral Pattern into Finite Automaton**

Method calls are the simplest syntactic element in a behavioral pattern. We transform a method call into two states and a transition between them, as shown in Figure 5(1). The symbol accepted by the transition consists of the concatenation of variables derived from the behavioral pattern: the caller's object, (a:A), a call symbol, "->", the callee's object, (b:B), and the called method, m1 (the latter two separated by a dot). The variables will be bound to concrete values incrementally during the dynamic analysis.

In the case of two subsequent method calls, such as the ones shown in 5(2), we translate both of them individually, in the way we just discussed, and then combine them by merging the end state of the NFA for the first call with the start state of the NFA for the second call.

A loop is also transformed into a start state, an end state, and a transition consuming the empty symbol $\varepsilon$. This transition is directed from the end state to the start state to repeat the loop. The inner elements of the loop are inserted between the two states. In the example in Figure 5(3), the sequence of two method calls is inserted and their start state and end state are merged with the states from the loop.

Using this approach, we transform the *State* behavioral pattern shown in Figure 4 into the NFA depicted in Figure 6. This NFA starts with an optional method call (states 0 and

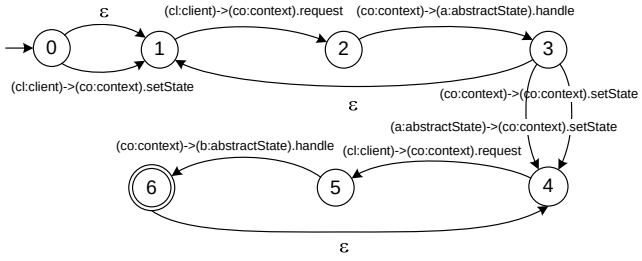


**Figure 6: NFA for the *State* Behavioral Pattern**

1). States 1 to 3 represent a loop that involves two method calls. The transitions between states 3 and 4 correspond to the alternative in the pattern, whereas the transitions between states 4 and 6 define a second loop. The accepting state of the NFA is the end state of the last element of the behavioral pattern (state 6). Note that an NFA might have multiple accepting states if, for example, the behavioral pattern ends with an optional fragment.

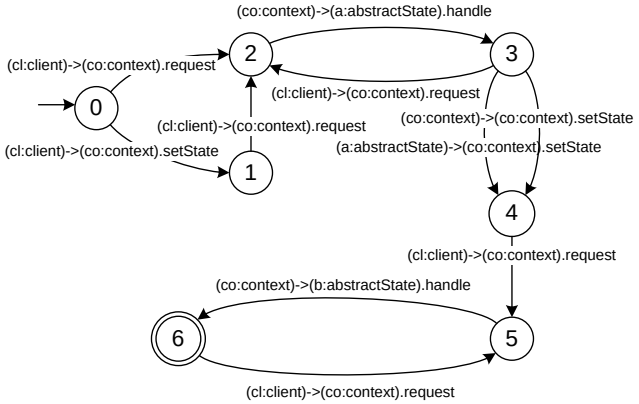


**Figure 7: DFA for the *State* Behavioral Pattern**

After constructing the NFA, the technique transforms it into a minimal DFA that is used by the dynamic analysis. Figure 7 shows the DFA for the *State* behavioral pattern.

So far, this DFA only accepts method call traces that conform to the behavioral pattern. However, behavioral patterns do not only describe valid method call traces, but also invalid ones. The methods mentioned in the behavioral pattern have to be called in exactly the order that is described by the pattern. Call traces in which such calls occur in any different order are invalid.

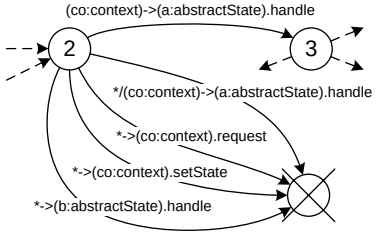


**Figure 8: Excerpt from DFA with Rejecting State**

For this reason, we introduce an explicit rejecting state (the crossed-out state in Figure 8) with no outgoing transitions. If a DFA reaches a rejecting state when processing a certain method call trace, that trace is invalid. To account for all possible invalid method traces, the rejecting state has incoming transitions from every state of the DFA (except for the start—it makes no sense to reject a trace consisting of a single call). For each state, these outgoing transitions consume method calls that are not allowed in that state.

To illustrate, Figure 8 shows the transitions that must be added to State 2 of the DFA in Figure 7. State 2 has an outgoing transition with the symbol (co:context)->(a:abstractState).handle, which indicates that this call is the only valid call from State 2 that involves the methods in the behavioral pattern. Therefore, all other calls involving such methods must be consumed by transitions going to the reject state. These calls are: calls to handle performed on a:abstractState by callers other than co:context (*/(co:context)->(a:abstractState).handle); calls to request and setState on co:context; calls to handle on b:abstractState. The callers are irrelevant for the latter calls because any call of these methods on the given objects is invalid at this point.

If for a method call event in a certain state there is no transition consuming it, the call event is ignored by the DFA. This means that, as mentioned in Section 3, any call to a method not mentioned in the behavioral pattern is ignored.

# 5. BEHAVIOR RECOGNITION

The purpose of the dynamic analysis is to monitor method calls of the program under analysis and match them against the automata derived from the behavioral patterns. The candidate patterns identified by the static analysis are used to instantiate the automata using the right mappings for class and method names. Figure 9 shows, as an example, a possible *State* design pattern candidate.

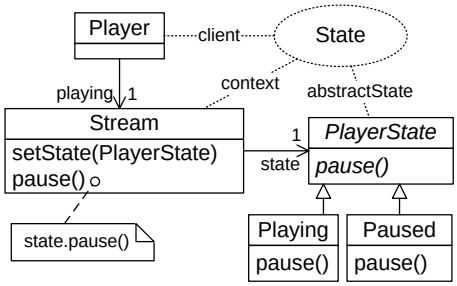


**Figure 9: A *State* Design Pattern Candidate**

The Player class has been identified as the client of the behavioral pattern. The Stream class has been identified as the context and the PlayerState class as the abstractState. The context.setState method of the behavioral pattern has been mapped to Stream.setState. The context.request method has been mapped to Stream.pause, whereas PlayerState.pause has been identified as abstractState.handle. Assume that we perform the dynamic analysis for this pattern and for the trace shown in Figure 10. We use this example to demonstrate how our technique matches the events in the trace against the automaton for the *State* behavioral pattern.

The dynamic analysis processes method-call events in two steps. In the **first step**, the analysis checks whether the call is a trigger for one of the behavioral patterns considered. A *trigger* is any method that labels a transition from the initial state of the pattern's automaton. If the call is a trigger for a behavioral pattern, an instance of the automaton for that pattern is instantiated, and the rest of the execution is checked for conformance with this behavioral pattern.

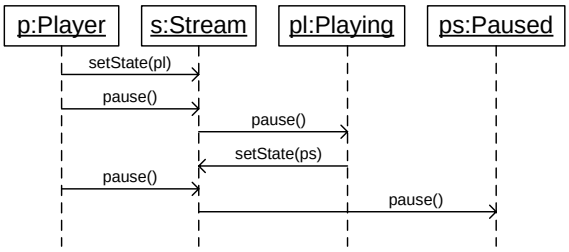


**Figure 10: Trace of the Program's Execution**

An automaton for a behavioral pattern may be instantiated multiple times during an execution (once for every time the automaton is triggered). Instantiating this potentially large number of automata may result in many instances of the various automata running concurrently, with obvious performance problems. To address this issue, we use a hybrid approach that combines Petri nets and finite automata. More specifically, we introduce tokens in the automata's states. An automaton may have an unlimited number of tokens in each of its states. Instead of instantiating a new automaton every time a trigger method is called, we simply add a token to the initial state of a single instance of the automaton.

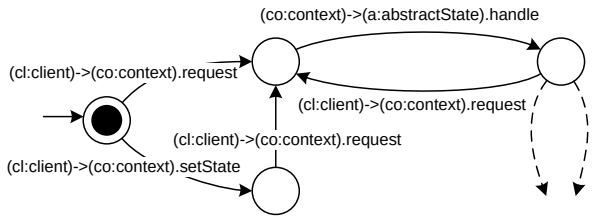


**Figure 11: Excerpt from an Automaton with Token**

| *Variable* | *Entity* | *Variable* | *Entity* |
|---|---|---|---|
| client | Player | cl | - |
| context | Stream | co | - |
| abstractState | PlayerState | a | - |
| setState | Stream.setState(PlayerState) | b | - |
| request | Stream.pause() | | |
| handle | PlayerState.pause() | | |

**Table 1: Initial Variable Binding of the Token**

The position of the token in the DFA indicates the state for that automaton's instance. In addition, the token encodes the bindings of the behavioral pattern variables (i.e., the class, method, and object names) to the entities identified by the static analysis (i.e., classes and methods) and dynamic analysis (i.e., objects). When adding a new token to the automaton, the token is initialized with the classes' and methods' mappings identified by static analysis, whereas the object variables are unbound (cf. Table 1). Figure 11 shows an excerpt of the automaton[1] for the *State* behavioral pattern with a token in its initial state.

In the **second step**, after triggering all necessary behavioral patterns, the method-call event is sent to all automata. For each token, the analysis checks whether there is an outgoing transition from the token's state that consumes the event. The analysis matches the method call event to the values of the symbol's variables. First, the event's caller and callee classes must match the ones specified in the transition's symbol (i.e., be the same class or a subclass of it). The called method matches if its signature is the same as the one specified in the mapping (i.e., it is the same method or a method that overrides it). If the caller and callee objects have already been bound, the objects involved in the current method call must be the same. Otherwise, the unbound object variable(s) is (are) bound to the caller and/or callee objects of the current method call event. If the method call event matches these criteria, the transition consumes it and moves the token to the target state of the transition.

[1]To improve readability, the rejecting state and all transitions to it are not represented in the figure.

| *Variable* | *Entity* | *Variable* | *Entity* |
|---|---|---|---|
| client | Player | cl | p |
| context | Stream | co | s |
| abstractState | PlayerState | a | - |
| setState | Stream.setState(PlayerState) | b | - |
| request | Stream.pause() | | |
| handle | PlayerState.pause() | | |

**Table 2: Variable Binding after applying Transition**

Table 2 shows the variable binding of the token after processing the first method call setState(pl) from p to s. Two of the four object variables are bound: cl to p, and co to s.

*Interpreting the Results.* When an execution terminates, all tokens are collected, and the result is calculated. Again, it is likely that multiple tokens have been created for each candidate because the candidate was instantiated and/or executed multiple times.

A token ends in a non-accepting, accepting or rejecting state. Because accepting states are allowed to have outgoing transitions, a token could have traversed an accepting state and terminate in a non-accepting or rejecting state. We therefore count the number of times a token traverses an accepting state, rather than simply counting how many tokens terminate in an accepting state. This number is an indicator of the likelihood for the candidate to correspond to the behavioral pattern considered. The higher the number of times a token traverses an accepting state, the longer the trace and the higher the confidence in the matching.

If a token ends in a non-accepting state without ever traversing an accepting state, we simply ignore it—such a token represents an incomplete trace and we have not enough evidence to decide whether the trace might have eventually got the token in an accepting or rejecting state. If a token ends in a rejecting state, we count the trace as evidence against the conformance to the behavioral pattern. If the same token traversed one or more accepting states before reaching the rejecting state, we still count the number of times the token traversed the accepting states. The rationale is that a part of the matched trace does conform to the behavioral pattern, and we want to take this fact into account.

At the end of the dynamic analysis, we report to the reverse engineers the ratio of the number of conforming traces to the number of non-conforming traces for each candidate design pattern. The reverse engineers can then decide whether they are satisfied with the results or they prefer to (further) adapt the pattern catalog and rerun the analysis.

# 6. PRELIMINARY EVALUATION

To demonstrate the feasibility of our approach, we performed a case study on parts of the ECLIPSE framework [4]. We used our current implementation of the approach that leverages the FUJABA TOOL SUITE [12] for the pattern speci-

fication and the static analysis. The dynamic analysis is only partially implemented yet, in that the event capturing and the runtime monitors are automated, but the transformation of the behavioral patterns into DFAs is mostly manual. Therefore, we generated DFAs only for two patterns: *State* and *Strategy*. As the basis for our case study, we chose three *Strategy* design pattern instances whose use in ECLIPSE is documented [5]. The first instance occurs in the Standard Widget Toolkit (SWT) and consists of a composite class that layouts its GUI components using various algorithms. The other two instances occur in a tree viewer in the *JFace* package and are used for sorting and filtering tree entries.

The static analysis recognized the three instances of the *Strategy* design pattern mentioned above. Because the structural patterns of *State* and *Strategy* are identical, the analysis also recognized them as *States*. Table 3 shows the context and the abstract strategy/state classes for these candidates.

| | | |
|---|---|---|
| Cand. 1 | Context | org.eclipse.swt.widgets.Composite |
| | Strategy/State | org.eclipse.swt.widgets.Layout |
| Cand. 2 | Context | org.eclipse.jface.viewers.StructuredViewer |
| | Strategy/State | org.eclipse.jface.viewers.ViewerSorter |
| Cand. 3 | Context | org.eclipse.jface.viewers.StructuredViewer |
| | Strategy/State | org.eclipse.jface.viewers.ViewerFilter |

**Table 3: Candidates identified by Static Analysis**

We executed ECLIPSE and captured method call traces. For this preliminary study, we simply started ECLIPSE and opened a project, and did not consider more complex executions. Table 4 shows the result of our dynamic analysis. For each candidate and behavioral pattern, two numbers are given. The former is the number of times a token traversed an accepting state; the latter is the number of rejected tokens.

| Behavioral Pattern | Candidate 1 | Candidate 2 | Candidate 3 |
|---|---|---|---|
| Strategy | 104/154 | 8/8 | 0/12 |
| Strategy (modified) | 502/14 | 110/0 | 92/10 |
| State | 0/1236 | 0/156 | 0/22 |

**Table 4: Result of the Dynamic Analysis**

For the study, we actually used two different versions of the *Strategy* behavioral pattern. We first performed the dynamic analysis with a version of the pattern that was directly derived from the pattern description [6]. The analysis correctly matched two of the three instances of the *Strategy* pattern. Candidate 3, however, was not confirmed. A manual inspection of the code revealed that the context class, namely StructuredViewer, references multiple strategies of type ViewerFilter. The request method of the viewer class iterates over all referenced ViewerFilters and calls their algorithm method, but the iteration is aborted as soon as one of the called methods returns false. This behavior deviates significantly from the descriptions of the *Strategy* design patterns in [6]. In fact, it corresponds more to a *Chain of Responsibility* than to a *Strategy* pattern.

By manually inspecting the rejected traces, we identified implementation variants of the *Strategy* behavior in ECLIPSE compared to [6]. For instance, the object setting the strategy and the object sending the request are different in contrast to [6]. We modified the *Strategy* behavioral pattern to reflect these variants. This customization shows the flexibility of our technique. Moreover, it is in line with our overall idea of an approach that allows for starting with a predefined pattern catalog and adapting the catalog to the actual context in which it is used. As expected, the dynamic analysis performed using this second version obtained better results for the first two candidates. Actually, it confirmed also the third candidate, even though we believe that it should not be considered an instance of the *Strategy* pattern. This shows that such customizations, although useful, must be used with care. Behavioral patterns that are too permissive could decrease the precision of the dynamic analysis in matching patterns and generate false positives.

As far as the results for the *State* candidates are concerned, the dynamic analysis did not match any of the traces with the *State* behavioral pattern. Therefore, it did not confirm any of the candidates identified by the static analysis.

Overall, we consider this case study very promising. The three spurious static analysis results (i.e., the three *State* pattern candidates) were not confirmed. Two of the three supposedly correct results (according to [5]) were confirmed. The third *Strategy* pattern candidate was not confirmed, but manual inspection revealed that the pattern does violate the pattern definition in [6]. Moreover, we have shown that the catalog could be easily customized to obtain better results.

# 7. RELATED WORK

Most existing techniques for detecting design patterns in source code are based on static analysis of structural aspects of the patterns only (e.g., [1, 8]). Because this paper focuses mostly on the detection of patterns based on dynamic behavior, we concentrate on approaches that perform design pattern detection using either dynamic analysis or a combination of static and dynamic analysis.

Heuzeroth et al. [7] combine static and dynamic analyses to detect design patterns. Static and dynamic patterns are specified as predicates using two Prolog-based languages, SanD-Prolog and SanD. The source code is represented in the form of predicates that encode its abstract syntax tree. The static analysis queries the source code representation based on the static specifications. The dynamic analysis checks the conformance of the program's runtime behavior with the dynamic specification, expressed as a state sequence. Although effective, this technique is limited by the use of the ad-hoc specification languages: SanD-Prolog is powerful, but specifications tend to be complicated and lengthy. SanD is more intuitive, but less powerful. For example, it cannot specify negative constraints.

Brown [3] also uses static and dynamic analyses to recognize design patterns. However, unlike our technique, Brown does not combine the two analyses. Patterns are detected using either static or dynamic analysis in isolation, which misses opportunities of leveraging the analyses' complementary strengths. Furthermore, the matching algorithms are hard-coded, which makes their maintenance difficult.

Kemmerer and Vigna present NETSTAT [13], a tool for network-based intrusion detection. Their approach models dynamic behavior by Labeled Transition Systems. The models encode intrusion scenarios based on network events. NETSTAT collects network events and matches them against the models to detect and prevent attacks in real time. Our approach is similar in spirit to NETSTAT, but targets a different problem and operates in a different context and under different constraints (e.g., overhead is not an issue for NETSTAT). In fact, we could probably leverage NETSTAT's infrastructure in our implementation.

# 8. CURRENT WORK AND CONCLUSIONS

We presented our technique for detecting design patterns by combining static and dynamic analysis. We also discussed a preliminary study that assesses the feasibility of the approach by applying it to a real, non-trivial application.

Although the study's results show that our approach can be effective, we are currently working on a more extensive evaluation. More precisely, we will apply our static analysis on the whole ECLIPSE framework and use the results for a dynamic analysis in the field. Our evaluation will leverage software tomography [2] to split monitoring tasks across many instances of the software and collect data by means of light-weight instrumentation. We expect this approach to work nicely because, in our context, monitoring tasks are split by construction—each design-pattern candidate naturally defines an independent monitoring task. A few of those tasks can be assigned to each deployed instance of ECLIPSE. When run in the field, these instances would then perform the dynamic analysis remotely and send back the results. Initially, we will have students in our research labs as users.

Another direction for future work is the use of our approach in different contexts. Our evaluation showed that this approach can not only recover design patterns, but also uncover "misuses" of design patterns with respect to their intended protocol. In this context, we are also investigating the application of our dynamic analysis in forward engineering. When designing interfaces for classes or components, behavioral patterns define a protocol of use for the interface. During the implementation phase, the program can then be tested for its compliance with those protocols.

Finally, from an implementation standpoint, we currently capture method call events using a tracing tool that leverages the debugging interface of the Java Virtual Machine [9]. This approach requires the execution of the software by the tracing tool which is inconvenient for evaluation in the field. To address this issue, we will change our implementation to gather method call events using instrumentation [11].

## Why Might the Approach Fail?

Some of the design patterns presented in [6] comprise too "little" behavior. For example, the *Adapter* or *Proxy* patterns just delegate a method call to another object. Using the dynamic analysis on such patterns would likely produce too many false positives. Our approach may not be applicable to patterns that have a too simple dynamic behavior.

The quality of our results depends on the representativeness of the considered execution sequences—the use of sequences that cover only a small fraction of the program's behavior would generally result in unreliable results. However, applying the approach in the field, on deployed software and on real executions, will help us address this issue. Moreover, finding a representative set of executions is a general problem for dynamic analysis, and our technique is affected by it like any other dynamic-analysis based technique.

We have not yet assessed the performance of the analysis in connection with the light-weight instrumentation and execution in the field. Based on our experience in instrumentation and data collection [11], we do not expect performance issues. If we do experience performance problems, we may have to trade space efficiency for time efficiency and analyze the collected event traces off-line.

## Acknowledgments

This work is part of the FINITE project funded by the German Research Foundation (DFG), project-no. SCHA 745/2-2. The work is also supported in part by NSF awards CCR-0205422 and CCR-0306372 to Georgia Tech.

# 9. REFERENCES

[1] G. Antoniol, R. Fiutem, and L. Christoforetti. Design Pattern Recovery in Object-Oriented Software. In *Proc. of the $6^{th}$ International Workshop on Program Comprehension*, pages 153–160. IEEE Computer Society Press, June 1998.

[2] J. Bowring, A. Orso, and M. J. Harrold. Monitoring Deployed Software Using Software Tomography. In *Proc. of the 2002 Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 2–9, Charleston, SC, USA, November 2002. ACM Press.

[3] K. Brown. Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk. Master's thesis, North Carolina State University, June 1996.

[4] Eclipse Foundation. *The Eclipse Platform. Online at http://www.eclipse.org*. Last visited: January 2006.

[5] E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. Addison-Wesley, Boston, MA, USA, 2003.

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

[7] D. Heuzeroth, S. Mandel, and W. Löwe. Generating Design Pattern Detectors from Pattern Specifications. In *Proc. of the $18^{th}$ IEEE International Conference on Automated Software Engineering*, pages 245–248, Montreal, Quebec, Canada, October 2003. IEEE Computer Society Press.

[8] C. Krämer and L. Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *Proc. of the $3^{rd}$ Working Conference on Reverse Engineering (WCRE)*, pages 208–215, Monterey, CA, USA, November 1996. IEEE Computer Society Press.

[9] M. Meyer and L. Wendehals. Selective Tracing for Dynamic Analyses. In *Proc. of the $1^{st}$ Workshop on Program Comprehension through Dynamic Analysis (PCODA) at $12^{th}$ WCRE, Pittsburgh, PA, USA*, volume 2005-12 of *Technical Report*, pages 33–37. Universiteit Antwerpen, Belgium, November 2005.

[10] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards Pattern-Based Design Recovery. In *Proc. of the $24^{th}$ International Conference on Software Engineering (ICSE), Orlando, FL, USA*, pages 338–348. ACM Press, May 2002.

[11] A. Seesing and A. Orso. InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. In *Proceedings of the eclipse Technology eXchange (eTX) Workshop at OOPSLA 2005*, pages 49–53, San Diego, CA, USA, October 2005.

[12] University of Paderborn, Germany. *Fujaba Tool Suite. Online at http://www.fujaba.de/*.

[13] G. Vigna and R. A. Kemmerer. NetSTAT: A Network-based Intrusion Detection Approach. In *Proc. of the $14^{th}$ Annual Computer Security Application Conference*, Scottsdale, AZ, USA, December 1998. IEEE Computer Society Press.

[14] L. Wendehals. Improving Design Pattern Instance Recognition by Dynamic Analysis. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, OR, USA*, pages 29–32, May 2003.

[15] L. Wendehals. Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams. In *Proc. of the $6^{th}$ Workshop Software Reengineering, Bad Honnef, Germany, Softwaretechnik-Trends*, volume 24/2, pages 63–64, May 2004.