Towards Pattern-Based Design Recovery

Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals
Software Engineering Group
Department of Mathematics and Computer Science
University of Paderborn
Warburger Straße 100, D-33098 Paderborn
Germany
[nierej, wilhelm, maroc, lowende]@uni-paderborn.de

Jim Welsh[1]
Software Verification
Research Centre
The University of Queensland
Australia 4072
jim@svrc.uq.edu.au

ABSTRACT

A method and a corresponding tool is described which assist design recovery and program understanding by recognising instances of design patterns semi-automatically. The approach taken is specifically designed to overcome the existing scalability problems caused by many design and implementation variants of design pattern instances. Our approach is based on a new recognition algorithm which works incrementally rather than trying to analyse a possibly large software system in one pass without any human intervention. The new algorithm exploits domain and context knowledge given by a reverse engineer and by a special underlying data structure, namely a special form of an annotated abstract syntax graph. A comparative and quantitative evaluation of applying the approach to the Java AWT and JGL libraries is also given.

1. INTRODUCTION

Past and current practice in many software producing companies means that good software engineering principles such as documentation, prior design and derived tests are less important than to be the first on the market with a new system or to produce quick solutions when problems with an existing system arise. Maintenance of the resultant systems quickly becomes the most difficult part of software development, particularly when the core developers have left the company.

In such cases, reverse engineering techniques then have to be applied to the existing source code to get a comprehensible, abstract representation and overview of the system as the basis for further enhancement. Typically, however, this reverse engineering is done partially and by hand, because the enhancements needed affect only part of the system and a complete reverse engineering task is too expensive. This approach leads eventually to a patchwork of code fragments and hot-fixes, and is more an aggravation than an improvement as far as the overall maintainability of the system is concerned.

To help avoid this patchwork effect, several reverse engineering tools for different reverse engineering approaches have been investigated. For example, tools exist for reverse engineering databases or applications written in various kinds of programming languages.

Design patterns are now seen as a logical basis for reverse engineering of (object-oriented) software systems - by recognising occurrences of known design patterns in source code, the implicit design may be recovered. Design patterns as originally presented by Gamma et al. [GHJV95] provide a collection of 'good' design principles together with a discussion of their advantages and disadvantages as well as their relation to other patterns. In the following, we call those patterns *Gang-of-Four* (GoF) patterns. GoF-patterns are the result of an intensive (more or less manual) reengineering process of existing software systems at IBM. Thus, GoF-patterns can be seen as a collection of recurring implementations made by independent developers.

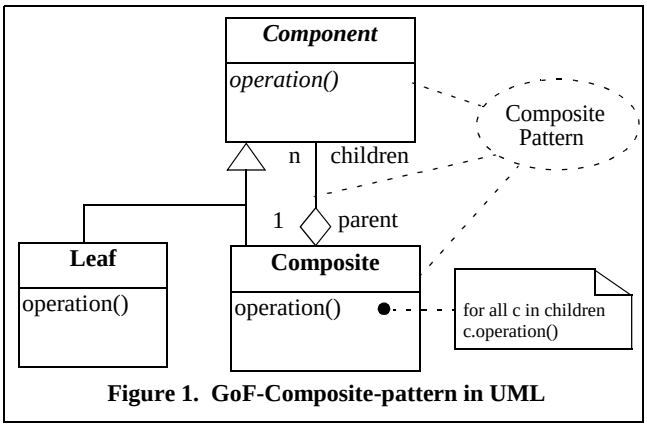

Figure 1. GoF-Composite-pattern in UML

Most parts of a GoF-pattern description are deliberately informal, to make the implementation (instantiation) of a pattern in an existing system as flexible as possible. The most formal part of a pattern description is the structure and sometimes the collaboration part. Gamma et al. used the Object Modelling Technique (OMT) to describe the structural part and interaction diagrams for sample collaborations between objects. Figure 1 shows the structure part of the GoF-pattern *Composite* using the Unified Modelling Language

This work is part of the FINITE project funded by the German Research Foundation (DFG) project-no. SCHA 745/2-1

Published in the Proceedings of the 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, pp. 241-251, ACM Press, June 2000.

1. Jim Welsh was a Guest Professor at the University of Paderborn in July and August 2001.

(UML). In addition to OMT's facilities, UML provides the facility to annotate parts of an object-oriented design as a pattern, shown as dotted lines from the annotated design parts to a dotted oval containing the name of the pattern.

In forward engineering, design patterns are a useful way to document certain parts of the design chosen, since they encapsulate information about the classes, methods and attributes used, and their relation to one another. The informality of their description presents no particular problem in this role. In contrast, reverse engineering tries to automatically generate such documentation from the available code, cf. [SS00]. In this context, design patterns are an equally logical way to express the knowledge discovered, but the flexibility inherent in their informal description becomes a problem for their automated recognition.

In practice, pattern-based approaches to design recovery have had limited success to date, in that only a small proportion of known design pattern instances are automatically recognised. This is mainly due to the facts that design patterns themselves often have a number of "slightly" different variants (cf. [SvG98]) and, more importantly, that many different implementation variants can be used for a chosen design pattern in the source code which are almost impossible to capture and to define in total. For example, Figure 2 shows three different implementations in Java that could all be used for the same structural relation in instances of the Composite pattern defined in Figure 1.

This variants problem manifests itself in two ways. Many tools that successfully detect design pattern instances in source code of larger systems confine their analysis to a level of granularity, e.g., call graphs and naming conventions for methods, that avoids the implementation variants problem. Typically, these tools produce many false positives which the reengineer then has to eliminate by performing more detailed analysis by hand, cf. [CFM93]. Other tools, which try to identify the behaviour of certain program parts by operating at a finer level of granularity, e.g., data flow and control flow graphs, fail to detect design pattern instances in larger systems because of the problems of scale that arise.

Our goal is to assist design recovery and code understanding by recognising instances of design patterns semi-automatically in source code and by annotating the code with the design parts that contribute to the pattern, as in Figure 1. The approach taken is specifically designed to overcome the problems of design and implementation variants outlined above. Section 2 further illustrates the problem of implementation variants by means of sample code. Section 3 then introduces our approach to formalizing patterns based on the abstract syntax graph of the source code and composing higher level patterns out of subpatterns to span the range of granularity required. Our execution algorithm and how it addresses the performance problems of fine-grained pattern recognition is presented in Section 4. Section 5 presents a comparison of our analysis results to date with those reported for alternative approaches. Section 6 summarises other related work not covered by the evaluation. Finally, current and future work are presented in Section 7.

2. THE VARIANTS PROBLEM

Detecting instances of design patterns (such as GoF-patterns) for design recovery requires that design patterns be formally defined, since informally described parts of the patterns are not amenable to (semi-)automatic recovery. The most formal part of design patterns is the structure part, which is typically expressed as a class diagram. Thus, Figure 1 shows the structure of the Composite pattern as an UML class diagram.

However, even class diagrams allow a developer to implement some aspects of the structure in many different ways. Assuming the system is implemented in an object-oriented language, it is reasonable to expect that UML classes are implemented as classes and that attributes and methods of a UML class become attributes and methods of the corresponding class in the source code. Likewise, inheritance relations in the UML class diagram should be directly mapped to the inheritance mechanism of the target language.

Besides others, more difficult parts are the structural relations between classes and their implementation in an object-oriented language. Typically, object-oriented languages do not have an explicit language construct to implement such relations, but assume the use of reference attributes as in Java or pointers as in C++. Figure 2 shows three different implementations for a so-called 'one to n' (1toN) structural relation between class *Panel* and *Item* in Java. Such an 1toN-relation is the basis for an aggregation, as required by the Composite pattern. Usually structural relations are mapped to reference attributes with appropriate access methods, but the access control and the container library used depend highly on the developer's experience, and/or on resource restrictions of the overall system. Variant 1 of the 1toN-relation implementation in Figure 2, for example, uses an array of type *Item* (line 3) to store all the item objects in a panel object and gives the developer full access to the array via access methods (line 5

```
 1: // Variant 1 (arrays with access methods)
 2: public class Panel
 3: {  private Item[] items;
 4:    ...
 5:    public void setItems (Item[] newValue)
 6:    { ...
 7:       this.items = newValue;
 8:    ...}
 9:    public Item[] getItems() { ... }
10: }
11: // Variant 2 (using container class library)
12: public class Panel
13: {  public HashSet items;
14: }
15: // Variant 3 (std. vector with access methods)
16: public class Panel
17: {  private Vector items = new Vector (100);
18:    ...
19:    public void addToItems (Item value)
20:    { ... }
21:    public void removeFromItems (Item value)
22:    { ... }
23:    public void draw()
24:    { ...
25:       Enumeration enum = items.elements();
26:       while (enum.hasMoreElements())
27:       { ((Item) enum.nextElement()).draw() }
28:       ...}
29: }
```

Figure 2. Different 1toN relation implementations in Java

and 9). Thereby, the developer has to deal with all array handling problems directly. Variant 2 is an implementation where the reference attribute is a container taken from a locally made container library with full access resulting from its public visibility. In contrast, variant 3 encapsulates the reference attribute (line 17) as a standard container (*Vector*) and provides appropriate access methods (lines 19 and 21). In addition, Variant 3's implementation of the *draw* method (line 23) indicates that the 1toN-relation is also an aggregation. This can be inferred from the implementation of the draw method, where a draw method call on a panel object is delegated to its contained item objects by the loop at lines 26 and 27.

These three example implementations of the structural relation between two classes are all consistent with the class diagram given for the Composite pattern in Figure 1, which is the most formal part of a design pattern description. In addition, many more implementation variants for an overall design pattern may arise, either from its less formally defined aspects or from a more liberal interpretation of the class diagram. For example, a developer may use a bi-directional (qualified) association according to good design principles, the inheritance relation may not exist or the Leaf class in Figure 1 may be omitted[1]. Whether these are viewed as implementation variants or 'slightly different' design variants depends on the rigour with which the design pattern description is interpreted. In either case, they contribute to the overall challenge of achieving effective recognition of pattern instances.

It should also be noted that many of the GoF-patterns are structurally identical and vary only in their behaviour, e.g., the *Strategy pattern* and *State pattern*. To distinguish between their instances it is obviously necessary to analyse behaviour, but this also increases the potential for more implementation variants and the need to deal with these effectively.

3. PATTERN DEFINITION

In our approach to pattern-based design recovery, patterns (and subpatterns) are defined with respect to the abstract syntax graph (ASG) representation of a program. Subpatterns define ASG structures that are constituent parts of other patterns or subpatterns.

As an example we apply our technique to the reverse engineering of Java programs. The approach presented is, however, not bound to any particular program language or any particular programming paradigm, such as the object-oriented paradigm, as will be explained later in this section.

The actual ASG used in our system is that produced by the JavaCC source code parser [JCC]. However, the ASG model presented in this paper is simplified compared to that produced by JavaCC for readability reasons. Figure 3 (without the oval-shaped nodes) shows an ASG as an UML-like object diagram, and its corresponding source code.

1. The collaboration description says: "If the recipient is a Leaf, then the request is handled directly.". This does not require the existence of an explicit Leaf class inheriting from Component. For example, such a design is used in the Java Swing API [ELW98].

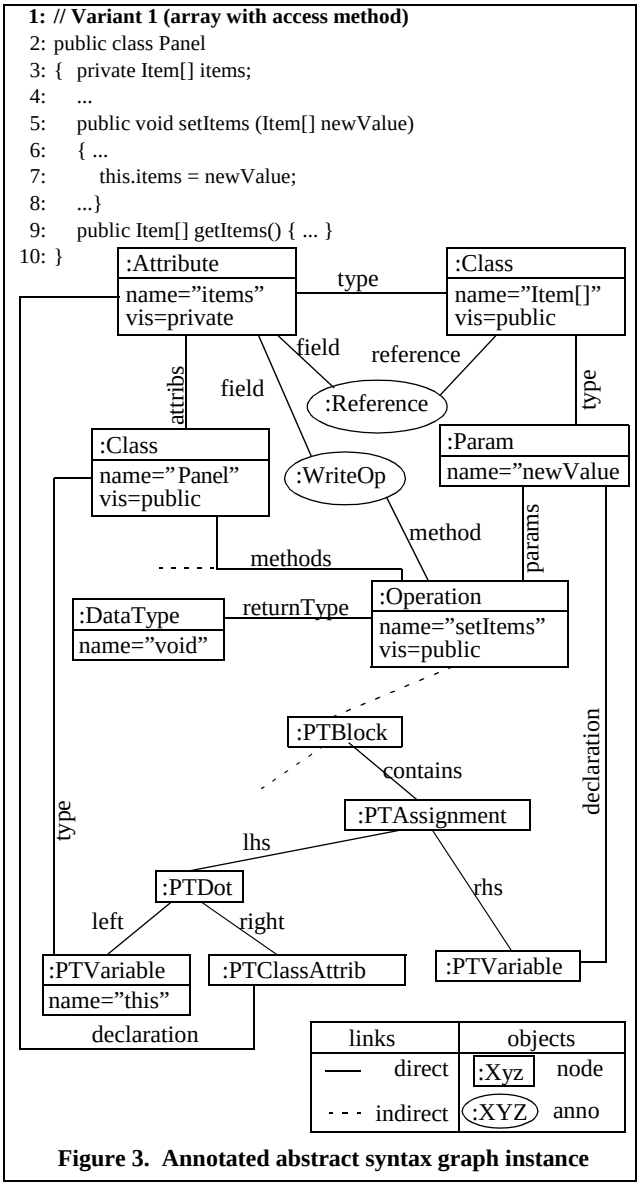

Figure 3. Annotated abstract syntax graph instance

Using an ASG representation has several advantages over using the textual source code representation. It avoids whitespace and formatting problems. It automatically normalizes the code for simple syntactic variants such as i=i+1 vs. i++. ASGs also provide additional information, such as identifier application and declaration links, that is useful in further analysis.

Recognising an instance of a pattern or subpattern in the ASG under analysis results in the addition of a corresponding annotation. The oval-shaped nodes shown in Figure 3 are annotations which identify certain subgraphs of the ASG as matching the named subpatterns. In the examples shown, attribute *items* is marked as a reference of the corresponding class, while method *setItem*s is marked as a write operation for attribute *items*.

Each pattern is formally defined by a graph transformation rule. The corresponding graph transformation annotates the ASG with additional nodes and edges (the oval-shaped nodes and corresponding edges in Figure 3) to indicate which subgraphs of an

ASG correspond to the pattern. Such subgraphs can then be used by rules defining other patterns that contain the defined pattern as a constituent part.

As a first example of such a subpattern definition, Figure 4 shows the transformation rule defining a subpattern which is an association relationship between two classes that each have an attribute annotated as a reference to the other class. In the notation

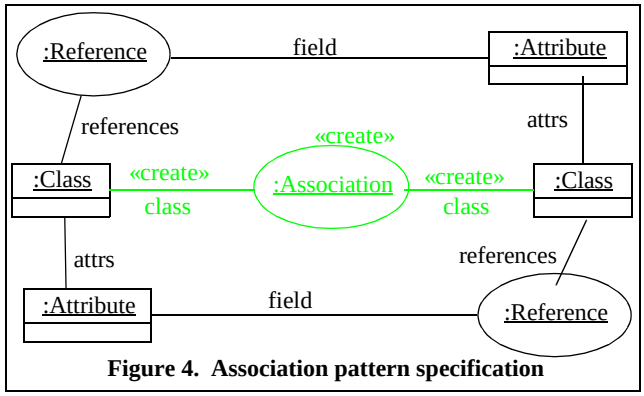

Figure 4. Association pattern specification

used, the subgraph to be matched in the host graph is defined by the black nodes and edges. The subgraph to be added is defined by the grey node(s) and edges annotated with the keyword “create“. This simple notation can be used because the rules only add information to the host graph and never delete any. (The formal definition and theory underlying such graph transformation rules is given in [SWZ95].)

The definition of a so-called *1N_Delegation* subpattern is shown in Figure 5. An 1N_Delegation requires the existence of a reference between two classes which involves a container class, i.e. an attribute definition in one class must be defined as a collection which contains objects of the type of the other class. The existence of that reference is given by the annotation *:ContainerReference*. In addition, a method body of that class (the *caller* class) must contain a call of an operation provided by the interface of the *callee* class. That call must appear within the body of a loop

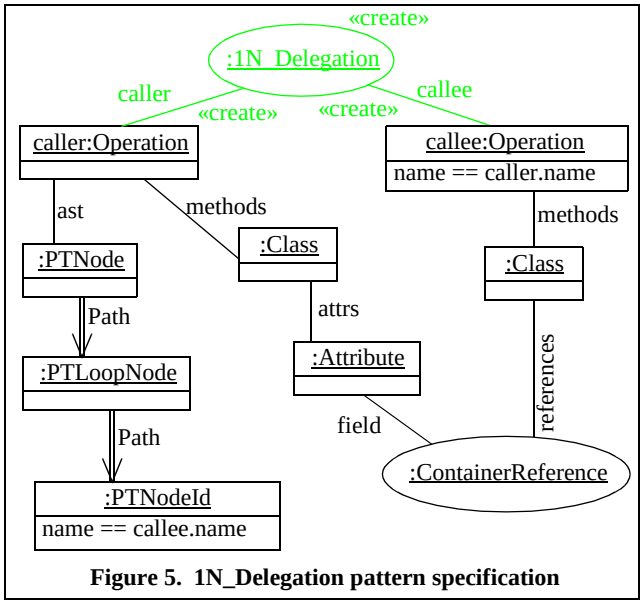

Figure 5. 1N_Delegation pattern specification

statement in order to support the assumption that the call is made to a particular item in a collection of items. Finally, the names of the called and provided operation must be the same. Each edge labelled *Path* in the definition indicates that an arbitrarily defined path in the tree part of the ASG must exist between the source and target node of that edge, i.e., the call can appear in an arbitrarily deep nesting of statements within a method body. This is a typical example of how many false positives can be avoided by checking method bodies in addition to type definitions in class headers. Furthermore, our definition of the *1N_Delegation* does not require application and declaration links between classes and objects, because this leads to very complex rules. In practice it is usually sufficient to identify a delegation only based on naming conventions and their corresponding appearance within method bodies.

Figure 6 then shows one possible definition of the *Composite* pattern. The definition requires that a generalisation and an association between the same two classes exist and that a Delegation pattern occurs between two operations of these classes. In effect, this definition describes the *Composite* variant without the existence of a leaf class (cf. Figure 1). Other variants require a slightly different definition.

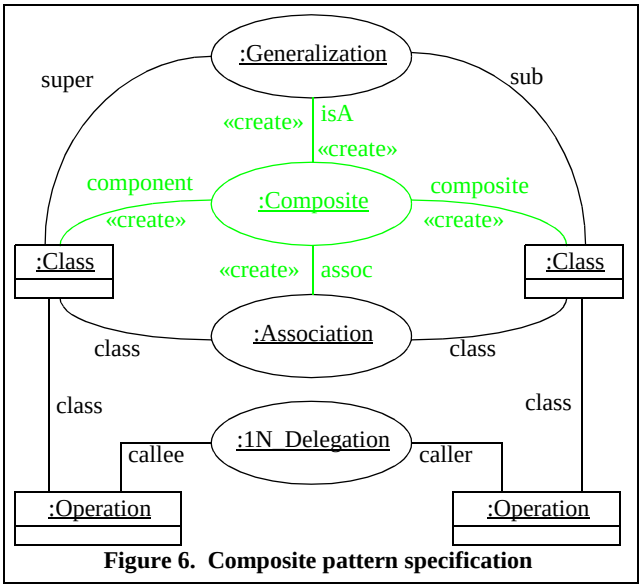

Figure 6. Composite pattern specification

The notation used here for pattern definitions is supported by the FUJABA[1] environment [FNTZ98, KNNZ99]. FUJABA supports (among others) the definition of UML class and collaboration diagrams and the definition of method behaviours as corresponding graph transformation rules which use the definitions of the class and object diagrams. The environment generates executable and complete Java code from these definitions. The graph transformation rules can be viewed as a subset of UML-like collaboration diagrams. They are drawn somewhat differently but can easily be translated into the UML syntax [KNNZ00].

1. The Fujaba (From UML to Java And Back Again) environment is developed by the Software Engineering Group at the University of Paderborn (www.fujaba.de).

In UML terms, the domain model for our pattern definitions is a class diagram defining the types of nodes and edges of an (annotated) ASG. The set of graph transformation rules are collaboration diagrams that define the annotations created to identify particular ASG patterns. As an example, Figure 7 gives an excerpt of the domain model (i.e., the class diagram) for our definition of patterns in Java source code. It identifies several subpatterns commonly used when constructing patterns. Associations in this diagram (not to be confused with the subpattern *Association* which is shown as a corresponding node in the diagram) are bidirectional, but with a defined read direction, which indicates for example that the *Composite* pattern annotates an *Association* and *Generalization* as well as two *Class* nodes of the abstract syntax graph (ASG). The default cardinality is exactly '1' in the read direction and 'n' in the reverse direction.

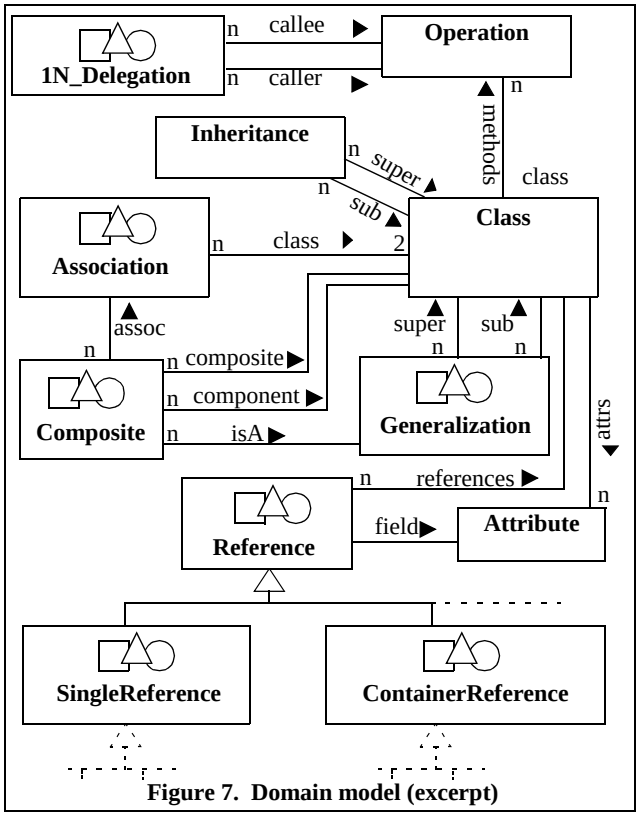

Figure 7. Domain model (excerpt)

As an example, the definition of a *Composite* as in Figure 6 can now be seen with respect to the domain model given in Figure 7. Following the UML syntax and style of definition, each object in this diagram must be an instance of a certain class in the domain model and each link between two objects must be an instance of an association in the domain model. In addition to the UML syntax, we require that the type names of the objects and the link names in the collaboration diagram must be the same as the corresponding class and association names. All these constraints are automatically checked by the FUJABA environment.

As is usual in UML-like definitions, the class diagram itself plays a constructive (as well as a consistency-checking) role in the overall definition of patterns. Inheritance relationships between patterns, as illustrated by the patterns *Reference*, *SingleReference* and *ContainerReference* in Figure 7, are defined solely by their representation in the class diagram.

In the domain model definition, classes representing patterns and subpatterns in the diagram are distinguished with stereotype 'Pattern' represented as an icon. Those without this icon represent ASG structures created directly by the parser from the Java source code. For these, there is obviously no need to create an annotation to express exactly the same information. This convenience reflects the fact that we are recovering an object-oriented model from an object-oriented language. To recover an object-oriented model from other languages, identifying a class may require a complex graph transformation rule as proposed e.g. in Rigi [MOTU93]. In principle, however, our approach is applicable to recovery of any kind of design model from any kind of programming language.

Using this approach, we have specified the GoF-patterns in a style similar to that shown for the *Composite* pattern. Doing so has shown first of all that GoF-patterns can have a precise definition, albeit with a judicious use of alternative variant definitions to accommodate the design flexibility intended. The definitions utilise a set of subpatterns similar to those shown in Figure 7 and illustrated by the *Association* and *1N_Delegation* patterns given in Figures 4 and 5. The problem of implementation variants impacts mainly on the definition of these subpatterns.

More significantly, our approach enables a reverse engineer to specify new design patterns or design variants in a flexible way, using a familiar UML-like notation. If he or she uses only subpatterns which have already been given a direct correspondence to code (such as *Reference*, *Association*, *Generalisation*, etc.), the engineer does not even need to worry about source code representations, but thinks only in terms of UML-like class, object and collaboration diagrams. Implementation variants accommodated by the existing subpatterns are automatically shared by each new pattern created. In effect, our construction of patterns from subpatterns exploits inheritance and use-relationships as in typical object-oriented specification languages. In general, using FUJABA, a reverse engineer can either adapt or extend an existing set of patterns such as those illustrated here, or build a complete new domain model for a programming language other than Java and for a target domain other than GoF-patterns. In the latter case, of course, he or she has to start from scratch by defining implementation variants and corresponding annotations as well as new design patterns.

Despite these benefits, our approach does yet not overcome the problem that each design or implementation variant, however slight, has to be defined explicitly at some level, if it is to be recognised. We revisit that problem in the final section of this paper to describe some current work designed to address it.

4. THE REVERSE ENGINEERING TOOL

Section 3 has described an effective formalism for defining a catalogue of patterns as the basis for design recovery from source code. The design recovery process for unknown systems is inevitably an iterative one. Typically, the reverse engineer first applies an initial set of patterns, then repeatedly examines the results, adjusts the patterns to address perceived deficiencies and reapplies them until a satisfactory outcome is achieved. To support this process the engineer needs a tool that applies the patterns to the source code involved and displays the results obtained.

To devise a tool that meets this requirement we adopt a threefold strategy. Firstly, we minimise the scalability problems mentioned in Section 1 by adopting the best available analysis algorithm. Secondly, we adapt this algorithm to deliver useful results incrementally rather than on completion. Thirdly, we involve the reverse engineer in the analysis process, to avoid unnecessary computation of unwanted analysis results.

4.1 The basic analysis algorithm

Pattern-based design recovery is a deductive analysis problem where patterns, or rules, are repeatedly applied to a representation of the source code to arrive at the most complete characterisation of the code permitted by the rules. Pure deductive analysis algorithms typically apply the rules involved level by level, bottom-up[1], according to their natural hierarchy, and produce useful results only when analysis is complete. Results from other researchers, such as [Wil96] and [Qui94], suggest that a reverse engineering tool providing fully automatic analysis based on this approach cannot scale for larger software systems.

Where patterns are defined as graph transformation rules, as in our case, graph transformation systems are the natural choice for implementing the tool. However, the scalability problem also applies to graph transformation systems such as Progres [Zün96] or AGG [AGG], which apply the rules in an arbitrary sequence usually determined by the internal data structures used.

FUJABA, in contrast to other graph transformation systems, only applies rules given a context, normally one object in the graph. While this is a restriction to the original theory of graph grammars, it has been shown not to be a problem in practical application. Its advantage is that it reduces the runtime complexity of the rule matching algorithm to polynomial size, whereas the original sub-graph matching problem is NP-complete [Meh84]. For more details we refer to [Zün96] and [FNTZ98].

By adopting FUJABA as the platform for our tool, we therefore reduce the problem of scalability compared to systems using standard approaches to deductive analysis. For the reverse engineer, however, this does not necessarily solve the performance problems involved.

4.2 Adapting the analysis algorithm

Although FUJABA reduces the computational complexity of analysis, a fully-automatic tool based on FUJABA is still undesirable, as the results are made available only when analysis is complete. Given that reverse engineering is an iterative process, such tool behaviour does not lead to an efficient overall process. Suppose, for example, our *1N_Delegation* pattern does not include the method body check shown in Figure 5. The resulting false positives manifest themselves early in the analysis, but the reverse engineer has to wait until analysis is complete to recognise them. For reverse engineering, therefore, a semi-automatic process is likely to be more effective, in which useful intermediate results are produced and the engineer is allowed to interact with them, either to add information and request that analysis continues or to revise the rule definitions and restart analysis.

To support such a process, the analysis algorithm itself must produce intermediate results useful to the engineer as early as possible, and be amenable to interruption and resumption without loss of results to date. Since the results most useful to the engineer are those produced by rules at the highest levels in the rule hierarchy, we adopt an analysis algorithm which combines a bottom-up strategy and a top-down strategy. Note that the algorithm affects only the execution sequence of patterns and does not violate their formalization as graph transformation rules.

To define the algorithm, the dependency hierarchy of the rules is levelled, such that each rule has a level number. A rule depending only on objects in the initial ASG gets number 1. A rule depending on other rules, i.e., whose definition includes annotations created by other rules, gets a higher number consistent with the natural topological order of the rules. Rules included in cycles concerning their dependencies get the same level number and are marked as recursive.

Figure 8 shows a snapshot of our analysis algorithm. The grey rectangle at the bottom represents all objects in the ASG. The black oval identifies an annotation already created by bottom-up analysis (with links to the objects annotated) while grey ovals represent a top-down analysis in progress. Directed arcs indicate the scheduling sequence of the rules. Variables at the arcs represent objects passed to the scheduled rule as context.

Bottom-up strategy

After parsing the source code to create the ASG, the analysis starts in bottom-up mode. Initially, all ASG objects schedule level 1 rules, i.e., those depending on ASG objects only. Scheduling only level 1 rules initially is sufficient to ensure that all necessary rule applications are eventually considered. It avoids many top-down failures that would otherwise occur, because the information available is not enough to establish a high level rule. Consider, for example, a *Composite* rule scheduled by a single class. The inherent search space is too large to justify its top-down investigation.

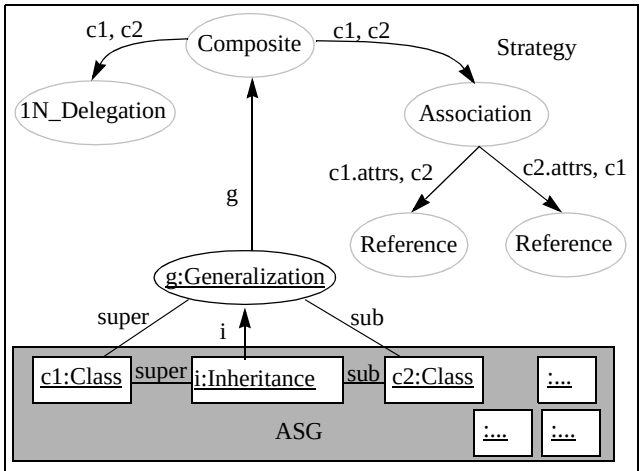

Figure 8. Sample analysis execution

1. In comparison, pure top-down approaches starting with top-level rules in the topology hierarchy are only of theoretical interest, because of the search-space implied. Even when a specific rule is identified for application, without an adequate starting context its top-down application is impractical.

An object *o* scheduling a rule *R* creates a rule/context pair *R(o)* which is added to a *bottom-up priority queue* held in *descending* order of rule level number. The use of rule level numbers to order the rule/context pairs in the bottom-up queue is not critical. Any ordering that promotes higher-level rules will do. This fact can be exploited to further tune the algorithm, as discussed in Section 7.

The algorithm continues in bottom-up mode by dequeueing the first rule/context pair, in our example *Generalization(i)*, cf. Figure 8. This rule is immediately applicable, so a generalization annotation *g* is created, annotating the superclass and subclass *c1* and *c2*, which are accessible via the inheritance object i, cf. the domain model in Figure 7. In contrast to ASG objects, which schedule level 1 rules only, creation of *g* schedules all rules depending on the *Generalization* rule, e.g., the top-level *Composite* rule.

Since *Composite* is a top-level rule, the pair *Composite(g)* is taken next from the bottom-up queue. At this point, however, *Composite*(*g)* cannot be applied successfully, since annotations have yet to be created by the other rules on which *Composite* depends (*1N_Delegation* and *Association*).

Top-down strategy

When a rule that depends on other rules cannot be applied in bottom-up mode, the algorithm switches to top-down mode, which uses a separate *top-down priority queue*. The top-down strategy tries to make the other rules create the missing annotations based on currently available information. In this case, the search space is quite strictly delimited by the information available, e.g., that inherent in the generalization *g*.

Consideration of *Composite(g)* in top-down mode thus schedules the *1N_Delegation* and the *Association* rules, cf. Figure 6. Where such rules depend on other rules, rule scheduling continues recursively. In our case, for example, the *Association* rule now schedules the *Reference* rule twice, as Figure 8 implies.

To establish a *Composite* consistent with *g*, the unique relevant context for both the *1N_Delegation* rule and the *Association* rule is the superclass *c1* and subclass *c2* obtained from *g*. In general, however, we note that alternative contexts may be implied for some rules, all of which have to be considered.

The rule/context pair at the front of the top-down mode queue is not dequeued if the rule involved schedules other lower-level rules. Instead, pairs added to the top-down queue are queued in *ascending* order of their level number. This means that the higher-level rule will be reconsidered after the lower-level rules on which it depends (if these succeed). Using a priority queue rather than a stack means that the top-down algorithm goes as far down the ASG as quickly as possible. This encourages earliest possible failure in top-down mode, while maintaining an appropriate sequence of rule applications for top-down success. If a rule marked as recursive is added to the top-down queue, however, stack behaviour is adopted until all rules so marked have been removed from the stack/queue.

When the rule at the front of the top-down queue can be applied, a corresponding annotation is created, all dependent rules are scheduled for bottom-up consideration, the front entry of the top-down queue is dequeued and the next element of the top-down queue is considered. The newly scheduled rules join the bottom-up queue since they represent analysis results that would have been created later in bottom-up mode anyway and need further investigation.

The algorithm runs in top-down mode until the top-down queue is empty or a rule in the queue fails with no alternative contexts left to explore. The first case means that the rule that started this top-down phase has been successfully applied, in our example the *Composite* rule. In the second case the starting rule cannot be applied in the given context. In either case the algorithm switches back to bottom-up mode.

Intermediate results

With the algorithm as described, each annotation once created represents an intermediate result that is not affected by subsequent analysis. In principle, therefore, the execution can be interrupted for inspection of results by the engineer at any stage. In practice, however, it is illogical to allow interruption during a top-down interlude, when some but not all of a closely related set of annotations may have been created.

Since the algorithm tries to establish high level rules using the top-down strategy, the intermediate results are likely to be useful information for the reverse engineer, e.g. GoF-patterns. The engineer can look at such patterns to determine if the analysis should continue on the current basis. The algorithm is also robust to certain changes by the engineer prior to resumption. Addition of annotations by the engineer is valid at this stage, provided these add all corresponding rule/context pair for dependent rules to the bottom-up queue. Marking a rule as 'to-be-deleted' is also acceptable, as the consequences of deletion can be systematically propagated to both the results to date and the resumed analysis. Such actions may be useful to the engineer as 'proofing actions' prior to permanent change to the rules themselves. Any addition or modification to the rules, however, invalidates the analysis to date and requires restart of the overall analysis.

The overall analysis finishes when the bottom-up queue is empty. In this case the algorithm has analysed all ASG objects and created annotations on the objects for all rules that could be applied.

4.3 Integration of the reverse engineer

Integrating the analysis algorithm described in section 4.2 into a semi-automatic reverse engineering process is easy because it is interruptible. Figure 9 shows our reverse engineering process as a statechart. The process starts by parsing the source code to create the ASG representation, followed by loading a particular pattern catalogue. The engineer can then make initial modifications before starting the analysis algorithm by sending a *start* event.

The complex state on the left-hand side with its two internal states '*bottom-up strategy*' and '*top-down strategy*' represents the analysis algorithm described above. The algorithm halts, and the reverse engineer can look at the results, if the algorithm has finished or the reverse engineer interrupts the execution by sending a *stop* event. As mentioned above, it is logical to confine such interruptions to bottom-up mode purely for pragmatic reasons.

The reverse engineer then has the opportunity to look at the results to see if the patterns selected still seem appropriate. By sending an

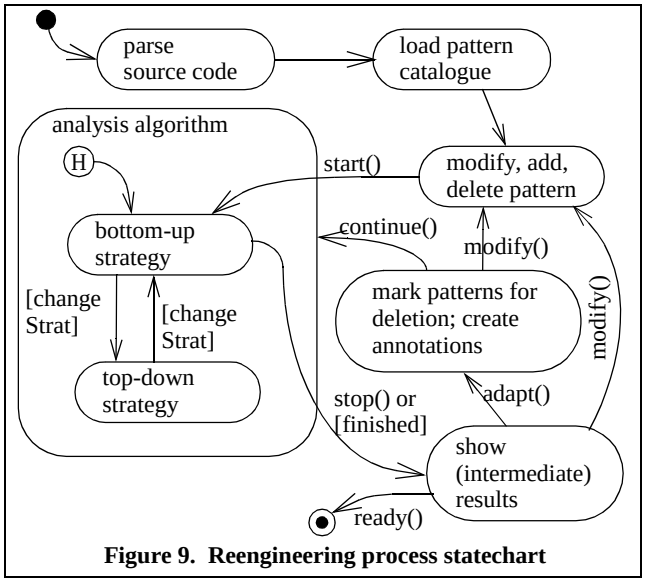

Figure 9. Reengineering process statechart

adapt event, he/she can mark patterns for deletion or create annotations that will steer the algorithm to a part of the source code that he/she wants to have analysed. Resuming rather than restarting the algorithm systematically propagates the consequences of such changes to both the prior and subsequent analysis. If the analysis to date fails to meet the engineer's needs in other ways, the patterns can be modified, but in this case the whole analysis must restart from the beginning.

5. EVALUATION

The tool outlined in section 4 has been implemented within FUJABA. To date, it has been evaluated on two major Java libraries, the Abstract Window Toolkit (AWT) [AWT] and the Java Generic Library (JGL) [JGL].

The AWT library is used to develop graphical user interfaces and provides graphical components such as buttons or text entry fields. Each graphical component is represented by a class in the library plus additional auxiliary classes. Overall, the library consists of 429 classes contained in 313 files totalling 114,431 lines of code. The AWT library is a good test-bed for pattern-based design recovery, because the developers have utilised GoF-patterns at many points in the source code. While this makes the analysis easier than a software system grown over several years of development, it makes the identification of false positives and the influence of the method body analysis easier to show. In addition, the results are comparable to those of other researchers who have used the same library, e.g. [SvG98].

As suggested earlier, a reverse engineer may start the analysis of an unknown system with a complete analysis using an existing pattern catalogue. For our first run we used a GoF-pattern catalogue that did not include any analysis of method bodies but only structural information. (In C++ terms, this equates to considering only header files.) This means for example that our 1N_Delegation pattern (cf. Figure 5) finds a delegation between two methods whenever two methods in two different classes have the same name. This is our first Java pattern catalogue JPC1.

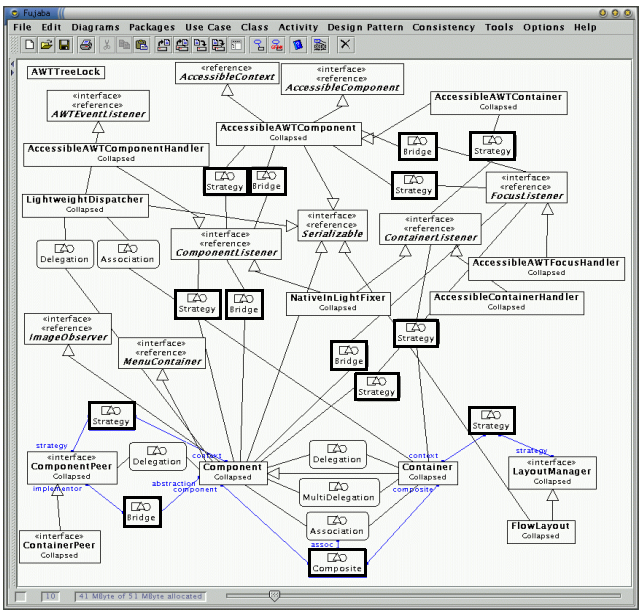

Figure 10. Results with JPC1 patterns

Figure 10 shows a screenshot of our reverse engineering tool after running this first complete analysis on the part of the AWT library (considered also by Seeman and von Gudenberg). The part is the central part of the library consisting of the *Component* and *Container* class and its connecting classes, comprising about 8,700 lines of code. The annotations corresponding to the 14 GoF-patterns found are highlighted. Checking the source code manually shows that four of the patterns found are real design pattern instances, the other ten are false positives and no other undetected design pattern instances exist.

Seeman and von Gudenberg [SvG98] found three of our four real pattern instances, namely a Composite pattern consisting of the *Component* and *Container* class, a Strategy pattern consisting of the *Container* and *LayoutManager* class and a Bridge pattern consisting of the *Component* and *ComponentPeer* class. They do not provide any information on false positives or performance issues, nor do they comment on the possible identification of our fourth real pattern instance, a Strategy pattern between *Component* and *ComponentPeer*.

This fourth pattern instance is not part of the results of Seeman and von Gudenberg because their precise definition of GoF-patterns does not identify a Strategy pattern as a part of a Bridge pattern. In our object-oriented definition of patterns it is natural to formalise the informal descriptions given by Gamma et al. in this way.

In addition to the false positive GoF-patterns, many false positive instances of subpatterns were also found. These are shown only partly in the screenshot, because of the lack of space. Note, hiding selected annotations is a basic functionality of the tool which offers the reverse engineer the opportunity to browse through the whole annotation structure more easily. For example, the engineer may initially be interested only in top-level patterns like GoF-patterns, but later wants to see subpatterns as well.

Similar results are presented from Krämer and Prechelt [KP96]. They only analyse the structural parts of a program, i.e. the header

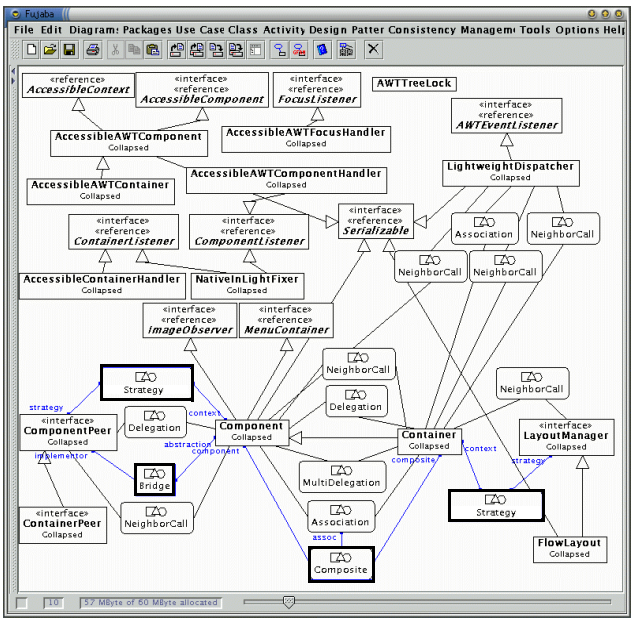

Figure 11. Results with JPC2 patterns

files in C++, which results in many false positives.

To improve on our initial analysis, we modify those patterns that could exploit analysis of method bodies. For example, we use the 1N_Delegation pattern definition as shown in Figure 5 and add or modify three other patterns that also analyse method bodies. This is our second Java pattern catalogue JPC2.

Figure 11 shows the analysis results of a complete run of the analysis algorithm using JPC2, focusing on the same constellation of classes as Figure 10. The false positive GoF-patterns have now disappeared and only real pattern instances are left, as in the results of Seeman and von Gudenberg. This result shows clearly that analysis of method bodies has a key role to play in reducing false positives.

In that sense our approach is similar to the approach of Seemann and von Gutenberg, i.e. they also analyse method bodies. Their approach, however, does not involve the reverse engineer and also does not support object-oriented construction and evolution by using a UML-like language as we do.

The impact of our object-oriented composition of patterns and subpatterns is reflected in the catalogue sizes that result. After their adaptation for analysis of the AWT library, both catalogues JPC1 and JPC2 contain 62 pattern definitions. This relatively low number results from the high reuse of definitions for subpatterns in the definitions for design patterns. For example, to define the Strategy, Bridge and Composite pattern only 15 subpattern definitions are used.

Seeman and von Gutenberg give no information on either false positive or scalability issues with their approach. To investigate residual scalability problems with our approach we then ran a complete analysis of the entire AWT library using the JPC2 catalogue. The complete analysis took approximately 22 minutes.

To verify the adequacy of our approach in other coding contexts, we also applied both catalogues JPC1 and JPC2 to a completely different source, the JGL. Because the JGL is more a collection of algorithms than a class library, the runtimes are faster than for AWT. Nevertheless, we obtained similar results than for the AWT, i.e., many false positives if we do not analyse method bodies.

Source code	KLOC	Patterns	Complete analysis	First GoF-pattern
AWT (part)	8.7	JPC1	41 sec.	0.5 sec.
AWT (part)	8.7	JPC2	100 sec.	56.0 sec.
AWT (all)	114.4	JPC2	1,307 sec.	13.0 sec.
JGL (all)	36.5	JPC1	43 sec.	3.5 sec.
JGL (all)	36.5	JPC2	73 sec.	24.0 sec.

Table 1: Analysis timing data[1]

As noted in Section 4, it is desirable that the reverse engineer can interrupt analysis at certain times to look at the annotations produced so far. Our tool also allows the engineer to request automatic interruption when certain patterns have been found, e.g., a Composite pattern. Using this facility, the time taken to find the first GoF-pattern was easily measured for each of the analyses described. As results in Table 1 show, the time to obtain the first GoF-pattern was often a small fraction of the overall analysis time in each case. This suggests that our top-down/bottom-up approach is well suited to supporting an iterative reverse engineering process.

The particularly early delivery of GoF-patterns with JPC1 reflects the detection of false positives, but is valuable, for example, in upgrading JPC1 to JPC2. It is such false positives that show up the inadequacy of JPC1's structure-only approach.

In Section 1 we identified scalability and false positives as the key problems to be overcome. On the evidence above, our adoption of FUJABA's graph transformation scheme has significantly addressed the scalability problem. Where previous researchers have been unable to analyse much more than a few thousand lines of code, our system deals comfortably with 100,000 lines of code. In addition, overall analysis times become less significant with the interactive analysis process that is enabled by early delivery of high-level annotations through our top-down matching algorithm. This combination of fast and interactive analysis appears to provide the basis for an efficient iterative reverse engineering process from the engineer's viewpoint.

Our system's capacity to analyse method bodies significantly reduces the false positives problem observed by other researchers who used only structural analysis. There is of course an additional analysis cost, since some pattern definitions are more complex and additional method-specific patterns are needed. As Table 1 shows, analysis of the central part of the AWT library takes 150% longer with JPC2 (which analyses method bodies) than with JPC1 (which

1. The times were taken while running on a Pentium III 933 MHz processor with 1 Gbyte of memory (with JDK 1.3 on Linux 2.4.5). All times include the runtime of the analysis algorithm only. Parsing times are not considered.

does structural analysis only). The corresponding increase for the analysis of the JGL is 70%. Overall, our experience suggests that the effort of taking method bodies into account is more than double that required for a structural analysis only.

6. RELATED WORK

Comparable work on reverse engineering of source code has been reported over the past decade. In [HN90] Harandi and Ning present program analysis based on an Event Base and a Plan Base. Rudimentary events are constructed from source code. Plans are used to define the correlation between one or more (incoming) events and they fire a new event which corresponds to the intention of the plan. Plans using events fired from other events is similar to defining patterns in terms of sub-patterns as presented in this paper. Each plan definition corresponds to exactly one implementation variant, which leads to a high number of definitions. This applies also to the approach of Paul and Prakash [PP94], where a matching algorithm for syntactic patterns based on a non-deterministic finite automaton is introduced.

An approach to recognize clichés, i.e., commonly used computational structures, is presented in [Wil96], within the GRASPR system. Legacy code to be examined is represented as flow graphs by GRASPR, clichés are encoded as an attributed graph grammar. The recognition of clichés is formulated as the sub-graph parsing problem which is NP-complete [Meh84].

Radermacher [Rad99] uses the graph rewrite system Progres [Zün96] to match patterns on the program. Patterns are defined as graph transformation rules and thus similar to ours. Radermacher uses the execution mechanism of the Progres environment, hence the execution is not incremental.

Analysing behaviour as well as structure using patterns is presented by Keller et al. in [KSRP99]. They use a common abstract syntax graph model for UML to represent the source code as well as the patterns. Matching the pattern's syntax graph on the program's syntax graph is done by scripts. These scripts are not generated automatically out of the patterns but have to be implemented by the reverse engineer manually. Descriptions in such a script language very quickly become large, awkward to read and difficult to maintain and reuse.

Besides offering no support for the user-friendly object-oriented construction of patterns, none of the approaches facilitate the exploitation of the human engineer's domain and context knowledge in refining the analysis required. This contributes to scalability problems for the process enabled.

Tonella and Antoniol [TA99] present an approach to recover 'coherent structures'. Their pattern definitions contain only quantitative statements, e.g. the number of classes, the number of inheritances, or the number of references. Hence, only structural information from the source code is used to identify patterns which are enriched with method call links afterwards. The approach does deliver metrics that indicate code quality. For the recovery of design patterns, however, it results in many false positives because of the quantitative definition of patterns.

7. CURRENT AND FUTURE WORK

Although the evaluation section has shown that our approach and the corresponding tool allows us to analyse large software systems and produces reasonable results, the algorithm offers several points for tuning and improving the tool.

One idea is to enable the engineer to control the priority definition for the bottom-up queue. This allows the engineer to place emphasis on particular patterns, resulting in the earliest possible analysis of those patterns. For example, he/she may wish to focus on certain patterns or to find out why a certain rule produces many false positives before continuing. Such priority definitions must be changeable to runtime, since the interests of the engineer change as analysis proceeds. Consequently, we are currently integrating a kind of analysis profile, which contains configuration parameters such as rule priorities for the algorithm, that can be switched during the analysis phase by the engineer.

As mentioned above, the reengineering tool utilises a set of basic subpatterns and patterns. We have defined a set of subpatterns that successfully recover (nearly) every instance of a GoF-pattern in our analysed software. However, the problem of implementation variants remains unsolved since we cannot ensure that we have defined all variants of subpatterns. Exploiting inheritance and use-relationships is not really a solution to this problem since each different variant still has to be defined explicitly.

To overcome the problem of numerous definitions of slightly different variants of subpatterns we are currently working on an approach where we replace several slightly different subpattern rules by one (more) general rule. Such a general rule, in principle, consists of the common parts of the different subpattern rules. To express the resulting impreciseness, we assign a fuzzy value to the resulting general rule.

The use of such general rules reduces the overall number of rules in comparison with defining each implementation variant by a separate rule. This reduces analysis time, but usually increases the number of false positives found during the analysis process. This results from the fact that a general rule may sometimes cover more instances than the corresponding individual rules do together. Therefore a balance between the number of general rules and the number of detected false positives has to be determined by the reverse engineer.

Our approach supports the reverse engineer in finding this balance by using fuzzy weights in a rule definition to define thresholds for accepting only those subpattern instances which have at least a certain precision. Finding a balance between the number of rules and false positives then reduces essentially to tuning the fuzzy values and thresholds. This can be done manually (which is part of our current work) or automatically, supported by a learning component (which is future work).

Future work will also seek to recover the architecture rather than just design fragments. In principle we can promote the compositional approach to recovery of GoF-patterns presented in this paper to a pattern language based architectural recovery. Furthermore, a resulting pattern algebra would also provide structural and behavioural inheritance of patterns.

REFERENCES

[AGG] Technical University of Berlin. *AGG, the Attributed Graph Grammar system. Online at http://www.tfs.cs.tu-berlin/agg*.

[AWT] SUN Microsystems. *AWT, the SUN Java Abstract Window Toolkit. Online at http://java.sun.com/products/jdk/awt*.

[CFM93] A. Cimitile, A.R. Fasolino, and P. Marascea. *Reuse Reengineering and Validation via Concept Assignment*. In Proc. of the 3rd International Conference on Software Maintenance (ICSM), pages 216–225. IEEE Computer Society Press, September 1993.

[ELW98] R. Eckstein, M. Loy, and D. Wood, editors. *Java Swing*. O'Reilly, 1998.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. *Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language*. In G. Engels and G.Rozenberg, editors, Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany, LNCS 1764. Springer Verlag, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[HN90] M. T. Hanrandi and J. Q. Ning. *Knowledge Based Program Analysis*. IEEE Transactions on Software Engineering, 7(1):74–81, IEEE Computer Society Press, 1990.

[JCC] SUN Microsystems. *JavaCC, the SUN Java Compiler Compiler. Online at http://www.suntest.com/JavaCC.*

[JGL] ObjectSpace, Inc. *JGL, the ObjectSpace (Voyager) Java Generic Library. Online at http://www.objectspace.com/products/voyager/libraries.asp*.

[KNNZ99] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. *Using UML as a visual programming language*. Technical Report tr-ri-99-205, University of Paderborn, Paderborn, Germany, August 1999.

[KNNZ00] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. *Integrating UML Diagrams for Production Control Systems*. In Proc. of the 22th International Conference on Software Engineering (ICSE), Limerick, Irland, pages 241–251. ACM Press, 2000.

[KP96] C. Krämer and L. Prechelt. *Design recovery by automated search for structural design patterns in object-oriented software*. In Proc. of the 3rd Working Conference on Reverse Engineering (WCRE), Monterey, CA, pages 208–215. IEEE Computer Society Press, November 1996.

[KSRP99] R.K. Keller, R. Schauer, S. Robitaille, and P. Page. *Pattern-Based Reverse-Engineering of Design Components*. In Proc. of the 21th International Conference on Software Engineering, Los Angeles, USA, pages 226–235. IEEE Computer Society Press, May 1999.

[Meh84] K. Mehlhorn. *Graph Algorithms and NP-Completeness*. Springer Verlag, 1st edition, 1984.

[MOTU93] H.A. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhl. *A Reverse Engineering Approach To Subsystem Structure Identification*. Journal of Software Maintenance, 5(4):181–204, John Wiley and Sons, Inc., December 1993.

[PP94] S. Paul and A. Prakash. *A Framework for Source Code Search Using Program Patterns*. IEEE Transactions on Software Engineering, 20(6):463–475, IEEE Computer Society Press, June 1994.

[Qui94] A. Quilici. *A Memory-Based Approach to Recognizing Programming Plans*. Communications of the ACM, 37(5):84–93, ACM Press, May 1994.

[Rad99] A. Radermacher. *Support for Design Patterns through Graph Transformation Tools*. In Proc. of International Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands, LNCS 1779. Springer Verlag, 1999.

[SS00] P. Selonen and T. Systä. *Scenario-Based Syntesis of Annotated Class Diagrams in UML*. In Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Minneapolis, Minnesota USA. IEEE Computer Society Press, October 2000.

[SvG98] J. Seemann and J.W. von Gudenberg. *Pattern-Based Design Recovery of Java Software*. ACM SIGSOFT Software Engineering Notes, 23(6), ACM Press, November 1998.

[SWZ95] A. Schürr, A.J. Winter, and A. Zündorf. *Graph Grammar Engineering with PROGRES*. In W. Schäfer, editor, Proc. of European Software Engineering Conference (ESEC/FSE), LNCS 989. Springer Verlag, 1995.

[TA99] P. Tonella and G. Antoniol. *Object Oriented Design Pattern Inference*. In Proc. of the 5th Symposium on Software Development Environments (SDE5), pages 230–238. IEEE Computer Society Press, September 1999.

[Wil96] L.M. Wills. *Using Attributed Flow Graph Parsing to Recognize Programs*. In Proc. of International Workshop on Graph Grammars and Their Application to Computer Science, LNCS 1073, Williamsburg, Virginia, 1994, November 1996. Springer Verlag.

[Zün96] A. Zündorf. *Graph Pattern Matching in PROGRES.* In Proc. of the 5th International Workshop on Graph-Grammars and their Application to Computer Science, LNCS 1073. Springer Verlag, 1996.