Understanding a System's Architecture

Dirk Heuzeroth

Universität Karlsruhe

IPD, Program Structures Group

Postfach 6980, 76128 Karlsruhe, Germany

heuzer@ipd.info.uni-karlsruhe.de

Welf Löwe

Växjö universitet

MSI, Software Tech. Group

351 95 Växjö, Sweden

Welf.Lowe@msi.vxu.se

April 1, 2002

Abstract

We present an approach to software visualization supporting the understanding of structure and behavior of software systems. To do so, we merge information from static program analysis with dynamic information obtained during the execution of the programs. The merged information is presented graphically in different views, where users can interactively choose between more abstract or more concrete ones. This technique is implemented in an analyzing tool; its open architecture enables the easy integration of new views such as metric or profiling views. Using these techniques, we detect components and interaction patterns in legacy code. We discuss our approach with the Observer Pattern as an example.

We evaluated our approach by self applying the tool looking for Observers in its code. We detect all Observer Pattern instances actually contained in the code as candidates, i. e., we do not miss any pattern instance. In the example, the candidates our tool considers Observers with high evidence include 80% of all actual Observer Patterns and no false positives.

1 INTRODUCTION

Understanding a legacy system's architecture is essential for the further development, the maintenance and the re-engineering of the system. Unfortunately, architecture is hardly documented in such systems; it must be retrieved from the system implementation using program analyses.

As these analyses are not unique by nature, system engineers habe to be involved to accept or reject certain results proposed by the automatic analyses. Hence, the result of such analyses ought to be presented in a form that is intuitive to the system engineer. Therefore, we provide multiple graphical views combining different aspects of the software to understand.

Views can be static or dynamic views: Static program information captures the program structure, but even elaborated analysis techniques obtain only little information on the runtime behavior of the program in advance. Hence, we additionally need dynamic information visualizing the behavior of an example run of the program.

The major task in understanding the architecture of a system is the identification of components and the essential communications between them.

Components are larger units of "coherent" modules or classes. The notion of coherency usually mixes static and dynamic system properties: it includes structural connection between the modules or classes in the call or inheritances graphs (static information). Additionally, it requires strong interactions between the modules or classes by procedure calls (dynamic information).

The essential communications between the components define the transfer of data independently of their implementations by simple calls, shared memory accesses, events, or callbacks. Many analyzes only picture this implementation of communication. E.g. static structure analysis often comes up with misleading results: Assume communication is implemented by an event-listener pattern. The source of the communication provides a method called by the target to add itself as an event listener. Note that the direction of this call is the opposite of the direction of the essential communication. Moreover, the event source captures the listeners in a container of abstract listener objects. There is usually no static type information pointing back to the communication target. This connection is only visible via the object identifiers captured in the communication source: Such information is runtime information.

As sketched above, both, components and essential communication between them, are defined by structure and behavior and, hence, require an understanding of static and dynamic system properties.

Combining graphic views of static and dynamic aspects of software and has already been established for software documentation. A typical means to illustrate programs are UML diagrams. There are a couple of different diagram types, such as static class diagrams, but also the dynamic sequence diagrams which

developers have found useful for communication. UML specifications show how a software system should behave, but not how it actually does.

For understanding large software systems, we have to reduce the amount of information displayed. Filters as well as aggregations perform such a reduction:

- Information filtering: users want to disregard parts of the software system or phases of the execution. They should be able to define, enable and disable filters.
- Information aggregation: Even if some information is analyzed, it is not necessarily displayed in detail. A single representative entity can visualize several computed objects and relations. Observers, e.g., are often interested in the coupling of classes instead of viewing each involved reference between them. The user should enable or disable the aggregation dynamically.

Filtering reduces the computed information and, hence, the run time of analyzes. However, if filtered information turns out necessary later, the static analysis must be performed again and the software must be restarted for the dynamic analysis.

Aggregated relations are always available. As aggregation is computed dynamically, it may be crucial for the analysis performance of large software systems.

We developed and implemented an architecture for the visualization of software structures and behavior. The VizzAnalyzer combines dynamic and static aspects of the software system using a front-end for the static analyzes, a debugger for dynamic analyzes and a visualization framework for the graphical presentation of the combined information. It allows for information filtering and aggregation.

Using this architecture as a basis, we analyze components and essential communications. We visualize the results of the analyses in a graphical form that is intuitive to the system engineer enabling to accept or reject these results.

The present paper describes this approach. It starts with a definition of our architectural model, which is the level of abstraction we strive to retrieve from the systems.

2 ARCHITECTURAL MODEL

For the purpose of this paper, we define components to be software artifacts with typed input and output ports. This definition focuses on computational components, but is sufficiently general to cover all other variants. Input ports are connected to output ports via communication channels called connectors. The notion of ports and connectors are known from architecture systems (Shaw and Graham 1996; Bass *et al.* 1998).

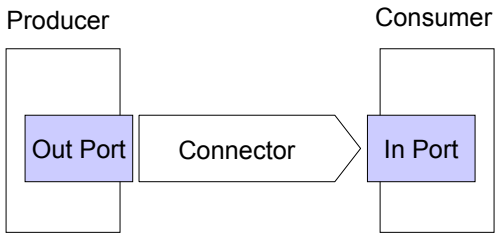

Figure 1: Basic Component Mode.

Some connectors may be as complex as most components, and thus require the same amount of consideration in design, but they all base on simple point-to-point data paths. Figure 1 sketches this basic component model.

In general, ports and connectors are implemented by patterns using basic communication constructs like calls, RPCs, RMIs, input output routines etc. provided by the implementation language or the component system. The Observer Pattern is such a port and connector implementation as it connects an event generator with some listener objects. The notification generally involves calling an event handling method of the listeners, where the subject waits for every call to return. Although, the pattern can be considered as asynchronous communication, since the events may occur arbitrarily, the notification itself constitutes a synchronous action.

In contrast to such an implementation, the ports and connectors themselves abstract from details. A port defines points in a component that provide data to its environment and require data from its environment, respectively. A connector defines out-port and in-port to be connected and specifies whether data is transported synchronously or asynchronously.

In order to extract components from a system and adapt them to a new environment, we prefer a view on the system containing abstract ports and connectors. However, legacy (source) code only contains port and connector *implementations* scattered throughout the code. The goal of our analyses is to compute the abstract port and connector view on these systems.

3 GENERAL APPROACH

We perform our analyses and visualizations to support system engineers in the understanding of the system architecture and behavior. Since the system architecture is almost always scarcely documented or even not available, discovering or recovering design information from existing systems is crucial for understanding and refactoring these systems. Therefore, tools to automatically extract design and architectural information are required.

We propose to retrieve static as well as dynamic information. Both are then combined to obtain the

desired information on the pattern to be detected. There are situations, where neither static nor dynamic analyses alone are sufficient (or not with acceptable expenses). E. g., it is not statically computable, which method or attribute is actually called or accessed at run time and how often. Even data flow analyses cannot predict all branches and loops, especially when the program to be analyzed requires user interactions. As objects are created at run time, relations among objects are dynamic by nature.

In order to get run time information of programs, we need a technique to mark, however, interesting program points and to get notified at run time whenever such a program point is executed. Furthermore, we need program state information with each notification. There are four alternatives:

Code instrumentation Code fragments are inserted at certain program points. The inserted code throws events at run time. In an initialization phase, every program point that is relevant for understanding the software is instrumented with such an event generation code. Together with the events, it provides information about the program state. A listener to these events could process and display this information.

Annotation of runtime environments provide an access to the program state at run time. They do not require code changes. The main draw back of this solution is its restriction to a certain environment version and implementation.

Post mortem analysis provide profiling information about the dynamic behavior of software systems. Interactions at run time are excluded.

Online debugger provides the same functionality as the annotation solutions. However, it does not require changes of the source code. Instead, it provides access to internal data structures of the compiled program. Additionally, it gives the infrastructure to load a program, execute it step by step and access the relevant program data in each step.

The Java Debug Architecture [6] provides a debugger interface for Java programs. It launches a run time environment for a pro-gram, starts a debugee and controls the execution of this debugee by another program. Additionally, the control program can access the state of the debugee's execution. Because of the free availability of the Java Debug Architecture, we decided to implement the tools in Java. However, approach and architecture are not restricted to this language. However, they require

- the source code to be available, and
- the programs to be executable to observe their dynamic interaction aspects.

We explicitly excluded all dependencies to coding and naming conventions. Hence, our approach also detects interaction patterns occurring by chance.

We distinguish static and a dynamic patterns. The former restricts the code structure the latter the runtime behavior. Analyzing with the static pattern results in a set of candidate instances in the code. In practice this set is large and programmers hardly want to screen all of them to detect the actual instances. Therefore, we test executions of the instance candidates found by the static analysis wrt. the dynamic pattern. Figure 2 illustrates our approach.

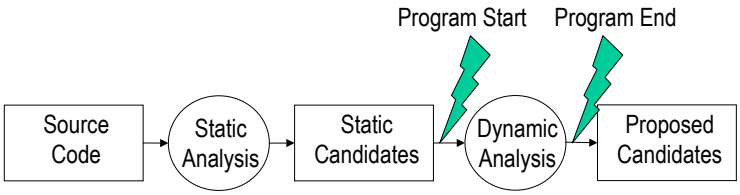

Figure 2: Process of detecting Communication Patterns.

The results of dynamic analyses depend on an execution of the candidate instances. Methods not executed at run time cannot be evaluated wrt. the dynamic pattern thus providing no information. However, testing techniques and environments guarantee that each reachable program part is executed while testing (of cause not every program sequence). Using these techniques, we may consider dynamic information available for each candidate instance. Moreover, we argue that parts that are less frequently executed are also less critical for understanding and for restructuring.

For the understanding of a system's architecture, intra-component communications do not matter. This provides us with the option to perform component analysis first and filter the candidates for communication pattern inside components leading to a tremendous reduction of communication pattern candidates. However, as our analyses are fast enough, we do not do so. Instead, we use the information on high level connectors in the code as an additional indication for the correct partitioning of the legacy system into its components. The hypothesis behind this is that components have should have an elaborated interaction interface to other components while intra-component interactions are often implemented ad hoc. Hence, we expect many sources and targets of essential communications reside on different components when choosing the right partitioning into components while many of the essential communications become intra-component for a not intended partitioning.

4 ANALYSIS AND VISUALIZATION TOOL BASIS

Designing views of software is not a trivial task: the static view requires static analysis of the software system usually done by compilers or similar tools. The dynamic view needs the execution of the software for some relevant input data. Additionally, we need some adequate graphic tools providing figures for the

static or dynamic information. Finally, some infrastructure should glue together the information gathering and the imaging devices. Our analysis tool, the VizzAnalyzer, uses two more general frameworks

- The Recoder for providing the static information on software systems,
- The VizzEditor for providing dynamic information and the actual visualization tools.

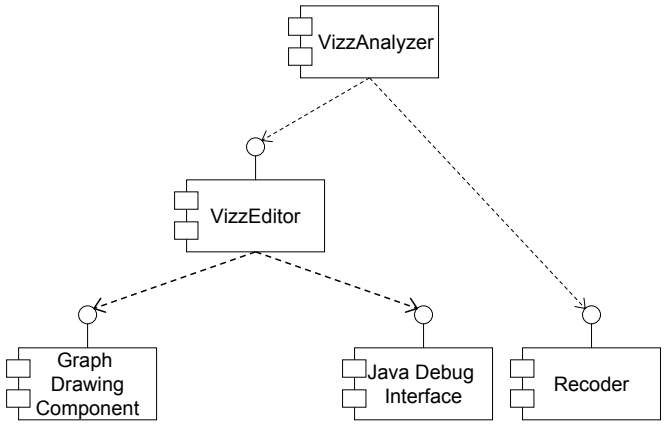

Figure 3: Components of the VizzAnalyzer.

The Recoder framework performs static analyses and program transformations. The former is used in our architecture; the latter is disregarded so far. Beside the standard implementations, users may integrate their own analysis algorithms and transformations.

The VizzEditor framework supports the rapid design of visualizations of general program runs. It maps information from a debugger interface to graphic views. Users may use the standard views or add special views appropriate for their programs. Figure 3 depicts the component diagram of the VizzAnalyzer.

The following subsections describe relevant aspects of this architecture. Subsection 4.1 defines the abstract information visualized, Subsection 4.2 shows our approach to deal with large systems, and Subsection 4.3 gives an idea of the information flow in the VizzAnalyzer.

4.1 Entities and Relations Displayed

For most object-oriented languages, entities of interest include methods, constructors, attributes, classes, packages, and objects. Objects occur at run time only, while the other entities are static by nature. Relations between them are also either static or dynamic. Table 1 gives an overview on the relations and distinguishes static and dynamic ones.

The relations are defined as follows: A method n **refers to** a method or attribute m, iff there is a (static) call statement or a (static) access operation to m in the body of n. A method n **calls** a method m or **accesses** an attribute m in a certain program run, iff the call statement or access operation to m in the

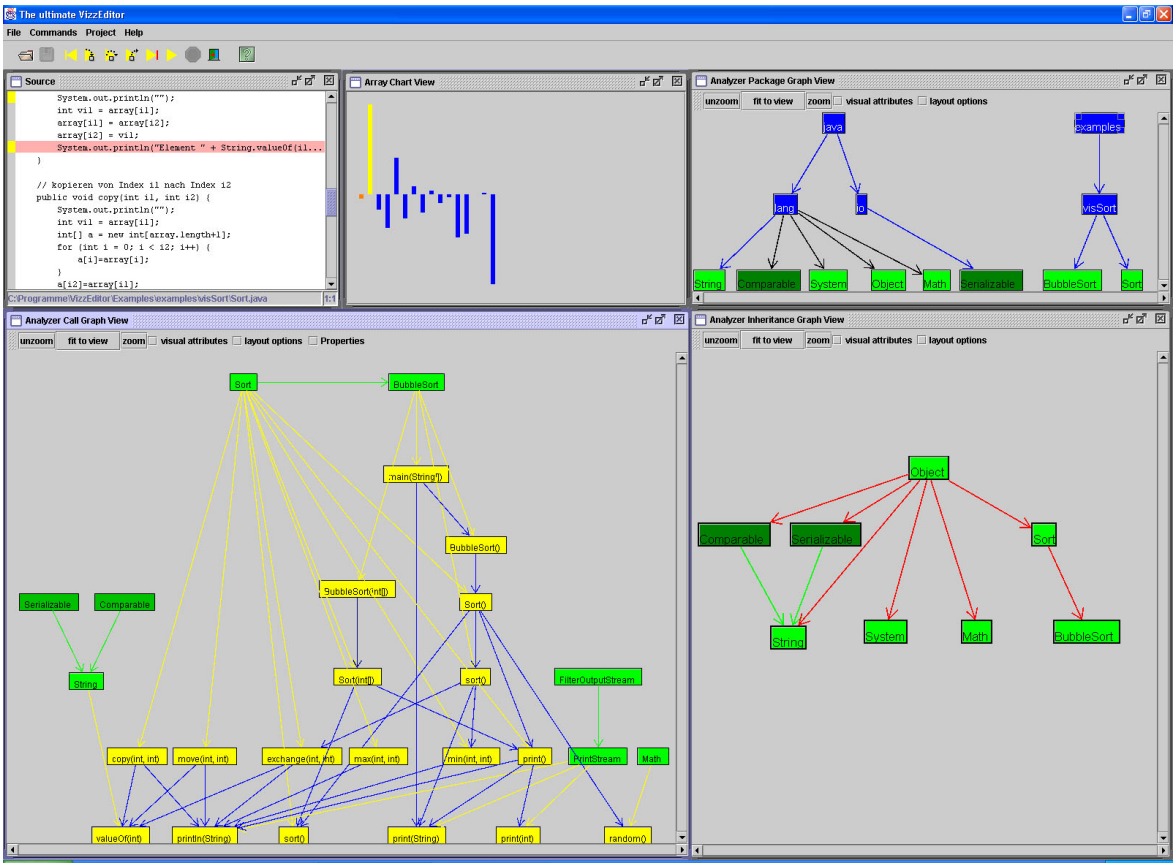

Figure 4: Visualization of a sorting program. Situation before program execution.

body of n is actually executed in that program run. The **contains** relation follows the static nestings of the corresponding entity definitions. An object x **knows** an objects y, iff an attribute of x contains a reference to y. The **subclass-of** relation is included in the static class definitions. An object x is an instance of a class Y iff its type attribute refers to Y.

The union of these entities and relations defines a graph with multiple node and edge types. Actually, there are three different graphs representing relations: the package graph, the inheritance graph, and the call graph. Except for dynamically loaded classes, the package and inheritance graphs show static properties, whereas the call graph merges static and dynamic information.

All graphs are displayed with automatic layout algorithms. Depending on the static or dynamic nature of their elements (entities and relations), the graph is computed at compile time and updated at run time, respectively.

Figure 4 depicts the state after static but before dynamic analyses. We use a simple Bubble Sort implementation as an example. The two frame on the top-left side show the program execution in a common

Static Relations	Dynamic Relations
Refers-To(method, method) Refers-To(method, attribute)	Calls(method, method) Accesses(method, attribute)
Contains (class, attribute) Contains (class, method) Contains (class, class) Contains (package, class)	Knows(object, object)
Subclass-Of (class, class)	Instance-Of (object, class)

Table 1: Relations Computed by the VizzAnalyzer.

debug view and the corresponding algorithm visualization (also produced by the VizzEditor). The top-right frame depict the package structure, the bottom-right frame the inheritance structure of the program. Finally, the bottom-left frame draws the Refers-To relation of the program. Packages, classes, interfaces and class members are easy to distinguish by their node colors, so are the different relations Refers-To, Contains, and Subclass-Of by their edge colors.

Except for trivial cases, it is not statically computable which method or attribute is actually called or accessed at run time and how often. Even sophisticated data flow analyses cannot predict all branches and loop iterations. Similarly, it is not statically computable which concrete method is called on a polymorph call. An additional problem is the detection of implicit method calls as they occur if a default constructor of a base class is called. As objects are created at run time, relations over objects are dynamic by nature. All these relations can only be computed dynamically for a concrete program run.

Despite of the variety of software systems, the data structures capturing the structure information and their graphical views are the same in all applications. Therefore, a fixed set of Recoder analyses can be applied. Moreover, potential program points of interest for the dynamic analyses are known in advance: we trace method calls, entries, exits, and attribute accesses. Constructors of objects are considered special methods. It is always clear how to update the graph views on the static analysis results at method calls, entry, exit, and attribute accesses events:

- Method calls and attribute accesses increase the weight of the edge corresponding to the respective Refers-To relation. Moreover, it may cause a redirection of the edges: Static analyses can only detect the static type of a callee. As the method called can be polymorphic, the actual call at runtime can be targeted to any of the callee's subtypes. It is therefore checked, whether the static and dynamic types of the callee are equal. Otherwise, the Refers-To edge is redirected to the method defined in the *dynamic* type of the callee.

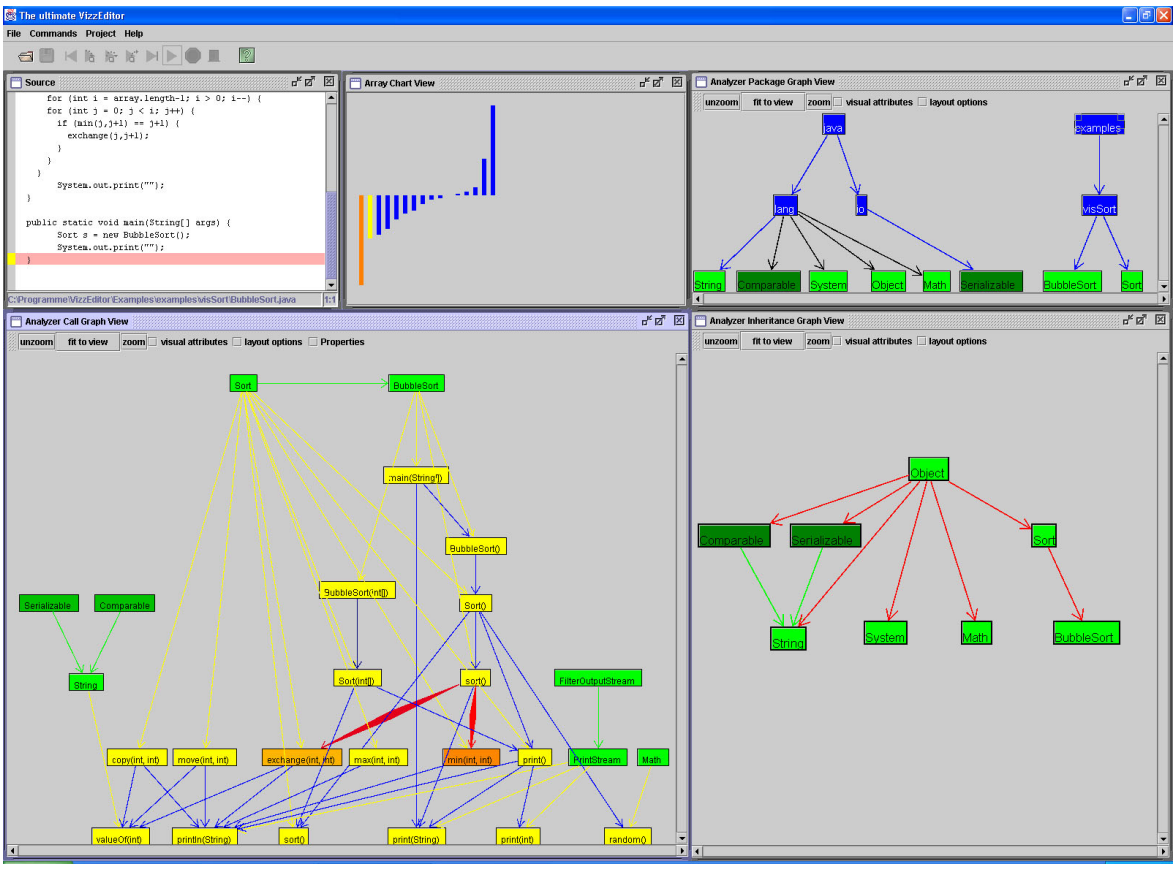

Figure 5: Visualization of a sorting program. Situation after program execution.

- A method entry increases the weight of the corresponding method node.
- An object creation increases the weight of the corresponding class node.

The weight of a graph node or edge can be related to its size or its color.

Figure 5 shows the example program after program execution (as the algorithm visualization frame indicates). As no class is dynamically loaded, package and inheritance graph remain unchanged. The call graph has been updated and indicates quite a few calls from the `sort` routine of the `BubbleSort` class to the `min` and the `exchange` routines defined in the abstract `Sort` class. On a color display, one can easily distinguish even the differences between the coloring of the `min` node compared to the `exchange` and their edges from the `sort` node. This indicates that the `min` routine has been called more often than the `exchange` routine (not surprising for an average run of the Bubble Sort algorithm).

4.2 Filtering and Aggregation

A graphic tool is only adequate if it allows abstraction and concretization of the information. Otherwise, it would fail in large applications. The call graphs depicted in the bottom left frame of Figures 4 and 5, respectively, show that already small examples might induce complex structures. In the VizzAnlyzer the reduction of information is performed by filtering and aggregation.

The granularity of filters is the package level. Users can exclude entities and relations from contributing to the displayed graphs by adding their package names to the static or dynamic filter list.

Whenever a class, a method or another entity is found by Recoder, its package name is tested using the static filter list. It is disregarded if it belongs to an excluded package.

If a call occurs at run time, the caller and callee classes are tested against the filter list. If caller or callee is already statically filtered, the call is not traced at all.

Additionally, the caller and callee packages are compared with the dynamic filters. Then the corresponding methods and classes are included in the static analysis but the call does not trigger dynamic changes of the views.

This feature allows the user to concentrate the analysis on the hot spots of the program and last but not least it speeds up execution.

Aggregation follows the subclass, containment and instance-of relations. A superclass may represent subclasses, a class its class members or objects. A problem occurs with multiple inheritances where, in general, the representative superclass is not unique. An easy solution is to forbid such aggregations. Alternatively, we can resolve the ambiguity by conventions: in Java, e.g., we would prefer the superclass as a representative to the super-interfaces. Finally, we could simply insert a copy of edges to aggregated classes to each superclass. Aggregation allows to define arbitrary levels of abstraction over the actual relations. The users may choose more abstract or more concrete views on their software systems.

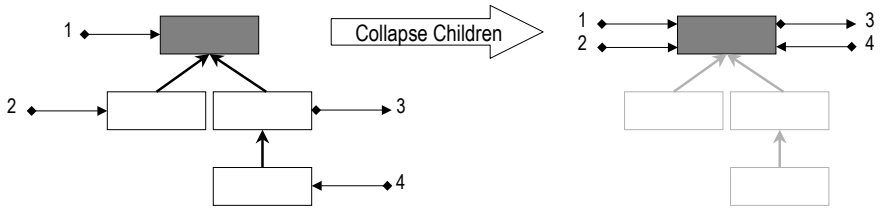

Figure 6: Aggregation of children: Original structure (left) and aggregated structure with nodes and edges actually disappearing displayed in grey.

Actually, we aggregate in both directions of the inheritance hierarchy: we can hide either the children or the parents of a class. If we collapse the children of a class X, we redirect to X all edges starting and ending in a transitive child of X. This is depicted in Figure 6.

If we collapse the parents of a class X, we redirect to X all edges starting and ending in a transitive

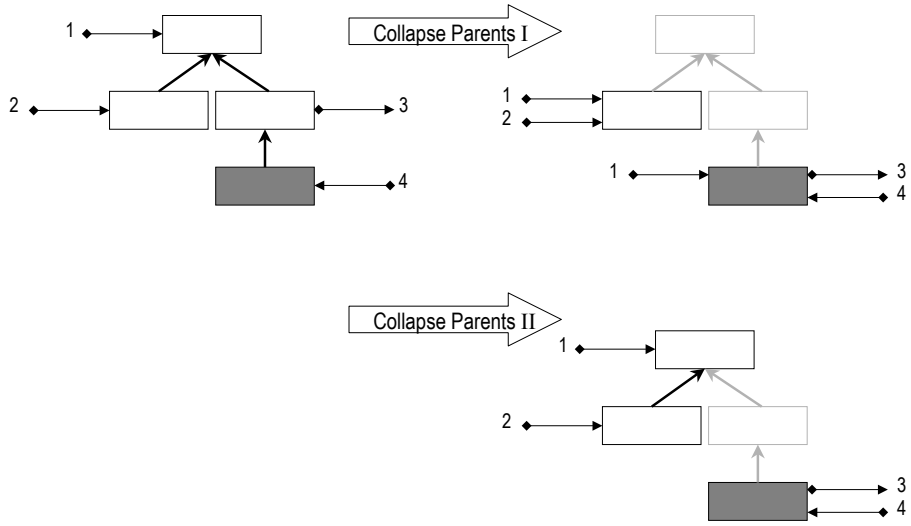

Figure 7: Aggregation of parents: Original structure (left) and aggregated structure with nodes and edges actually disappearing displayed in grey, (I) with edge copying and (II) without.

parent of X. For parents with more than one children, we have two options:

- As edges starting and ending in the parent node to disappear cannot uniquely assigned to either of the children, we copy those edges and assign it to all children.
- Collapsing parents could stops at the first transitive super class, which has more then one child. Then we do not copy edges.

These two options are depicted in Figure 7.

In addition to the aggregations along the relations defined by the program, users can define custom relations for aggregation. For optimizing a package structure with respect to the communication frequency internal communications are maximized while communications to other packages are minimized. Then, it is desirable to define new, temporary relationships modelling possible packages. The effects of each "virtual" package structure are visualized before performing the corresponding reorganizations at hand.

Figure 8 depicts the program execution in the same state as Figure 5. In contrast to the drawing of the call graph in Figure 5, all class members are aggregated. The `sort` routine of the `BubbleSort` class extensively called the `min` and the `exchange` routines defined in the abstract `Sort` class. Hence, the edges between the method nodes are propagated to the respective parent nodes. This makes the graph much more concise.

4.3 Information Flow in the System

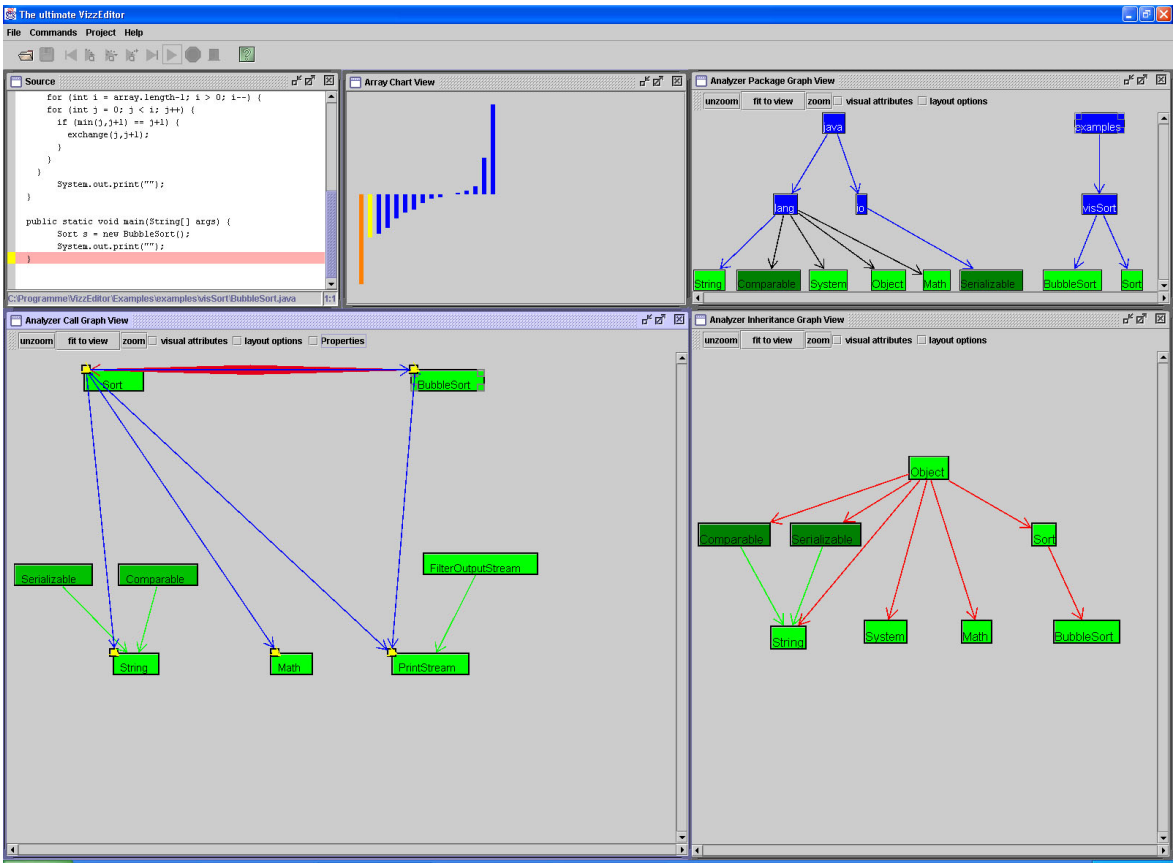

Figure 8: Aggregated situation after program execution: all class member, i.e. children of class nodes, collapsed.

The program to analyze is first loaded and analyzed with Recoder, cf. Figure 9 top. In principle, Recoder is an compiler front end. It reads the source code and constructs the syntax tree. Then it performs the static semantic analyses where it resolves names and types. In addition to standard semantic analyses, it can compute additional relations over the syntactical and semantic elements, e.g. define-use-relations. It has a programmer interface to add new, user defined analyses for the computation of arbitrary relations. In contrast to standard compilers, Recoder provides these entities and relations at another interface. This interface is accessed by the VizzEditor.

The VizzEditor accesses the Recoder Open API and extracts the static relations given in Table 1. They are transformed into models of graphical views. These models contain all elements drawn by the actual views but abstract from some visual properties, like color and layout algorithms for graphs. In general there is a $1 : n$ relation between a view model and its views.

The static relations are visualized with the respective graphs, cf. Figure 9 middle. Graph drawing is

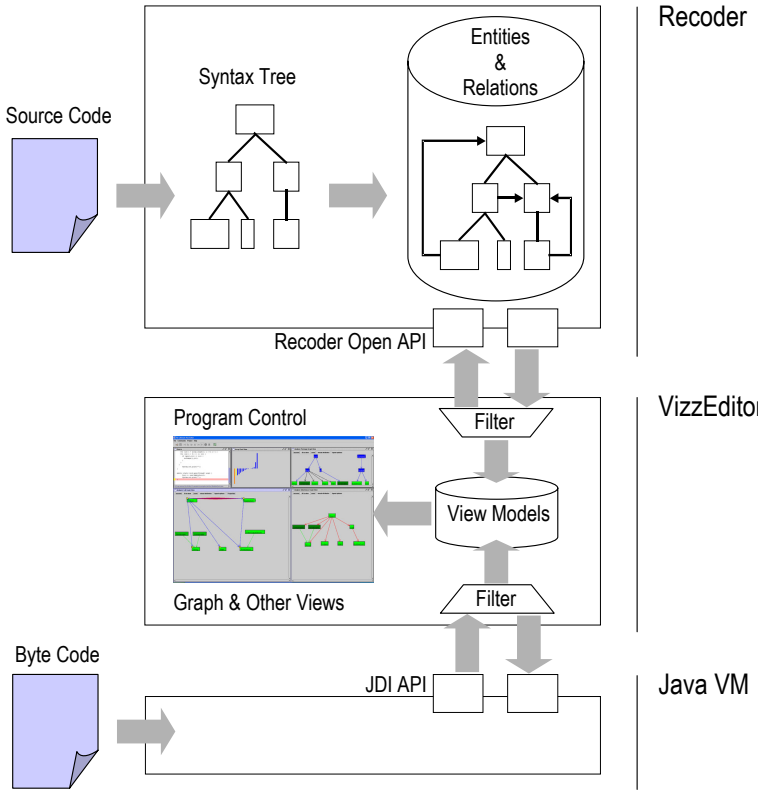

Figure 9: Information Flow in the System.

done automatically using one of the predefined layout algorithms. These algorithms include upward drawings and spring embedding algorithms. The user also may stop the automatic layout and rearrange the nodes interactively. The VizzEditor has also a programmer interface to add new, user defined layout algorithms. With the graphs visualizing the static relations drawn, the program can be executed.

Program execution is done in a standard Java Virtual Machine (VM). However, the executions is controlled via the Java Debug Interface (JDI) provided by the Java VM. Whenever a relevant program point (method call, exit, entry, attribute access) is executed, the VizzEditor receives an event through the Java JDI API, cf. Figure 9 bottom. If no filter applies, the corresponding view model and, with it, the graph displayed are updated accordingly, cf. Subsection 4.1.

The user may stop the execution of a program to inspect arbitrary program states in more detail. While the visualizations are frozen, the user can exploit the usual debugger functionality also provided by the VizzEditor. The current position in the program execution is not only marked in the source code view but also in the graphical views: the node representing the current program position is marked. Additionally, the user obtains information on the nodes (representing packages, classes, objects, or methods) and edges (representing relations) displayed in the views. The user may resume the program execution or proceed step-by-step.

Usually, the Recoder is only busy in the initialization phase. Howevwer, if a new class is dynamically loaded, it is passed to the Recoder. Its package is added to the package structure; the inheritance, the reference and the call relations are updated as well. This triggers an update of the view models and thereby and redrawing of the graph views.

5 UNDERSTANDING THE COMPONENTS

To demonstrate the practical results, we analyzed the Recoder system itself. As Recoder contains more then 600 classes and 80.000 lines of code, this example shows the ability to handle with large systems. Figure 3 shows the call structure of Recoder. First filtering reduces the amount of classes - we filtered utility and system classes.

Further reduction is due to aggregation: we collapsed all methods and a number of sub- and super-classes. As classes and interfaces as well as normal and constructor calls are colored differently, the graph appears quite clear.

Figure 4 shows a zoom into the call graph. Here, we displayed the methods. We observe heavy dynamic interactions in one part of the program indicated by the fat call edge (calls from the parser to the lexer - a well known hot spot for compiler front-ends), while other parts are not active so far.

6 UNDERSTANDING THE CONNECTORS

In this section we present our approach to detect interaction patterns by combining static and dynamic analyses. Subsection 6.1 discusses the static analysis, Subsection 6.2 the dynamic analysis.

In the remainder of the document, we consider the Observer Pattern (Gamma *et al.* 1995) (event notification) as a special architectural pattern. It is frequently used in frameworks and applications to realize loose coupling of objects or components. Suppose the following scenario: we tailor the framework or application for an environment requiring efficient communication among statically known partners. In such a setting, the Observer Pattern would be inappropriate. Thus, we need to detect it and replace it by a more efficient solution.

We need static and dynamic analyses to detect the Observer Pattern. The static analysis computes a set of classes that fulfill the necessary properties for subject and corresponding listener classes. The dynamic analysis then monitors objects of these classes during execution and checks whether the interaction among them satisfies the dynamic Observer protocol.

Implementations of ports and connectors follow communication design patterns. In order to retrieve

```
public class Subject {
  private Container c = new Container();
  private State s = new State();

  public void addListener(Listener l) {
    c.add(l);
  }
  public void removeListener(Listener l) {
    c.remove(l);
  }
  public void notify() {
    if (s.notChanged()) return;
    for each l in c:  l.update(s);
  }
}
```

```
interface Listener {
  public void update(Object o);
}

public class MyListener
  implements Listener {

  public void update(Object o) {
    doSomething(o);
  }
}
```

Figure 10: Pseudo code snippets sketching an implementation of the Observer Pattern

an abstract view, we search for the patterns. The static analysis computes potential program parts playing a certain role in a communication pattern. The dynamic analysis further examines those candidates. We can thus consider static and dynamic analyses as filters that narrow the set of candidates in two steps.

In the subsequent subsections, we use the Observer Pattern as a running example. The following naming conventions refer to roles of certain methods of the Observer Pattern. Figure 10 sketches an implementation. Note that this naming convention is only used for explanations in this paper; the static analysis does not refer to those name.

addListener**:** a method responsible for adding listener objects to a subject object.

removeListener**:** a method responsible for removing listener objects from a subject object.

notify**:** a method responsible for notifying the listeners of a state change in the subject.

update**:** a method implemented by the listener objects, called by the notify method.

We assume that addListener, removeListener as well as notify are contained in a single class and are not distributed among different hierarchies. This is not an unnatural restriction, but reflects object-oriented design principles.

6.1 Static Analysis

The program source code is the basis for the static analysis; it is represented by an attributed abstract syntax tree (AST) as computed by common compilers. A *static pattern* is a relation over AST node objects. It is defined by a predicate P using the information in the attributed AST as axioms. Names of variables, methods and classes nodes may be compared with each other but not with constants, thus making the pattern definitions independent on naming conventions.

The static analysis reads the sources of the program in question and constructs an attributed AST. Then, it computes the pattern P relation on the AST nodes and provides the result as a set of Candidates, i. e., pattern instances with the appropriate static structure. This set is a conservative approximation to the actual patterns in the code. The dynamic analysis, cf. Section 6.2, refines this approximation later on.

An Observer Pattern candidate is a tuple of method declarations of the form:

$$(\mathsf{S.addListener}, \mathsf{S.removeListener}, \mathsf{S.notify}, \mathsf{L.update})$$

where S is the class declaration of the subject of observation and L the class or interface declaration of the corresponding listeners.

In practice, the candidate set is large. Brute force methods, e. g. Prolog like resolution, are therefore not appropriate for use in practical tools. The search should be more directed.

To produce the candidate set for our example, the static analysis iterates over all program classes and their methods. For each method m of a class c, we first assume it plays the addListener or removeListener role. Therefore, we consider each parameter type p of method m a potential listener, provided p is neither identical to nor a super or a subclass of class c ($\mathsf{p} \not\succ \mathsf{c}$). Such a relation would contradict the decoupling of subject and listeners as defined in the Observer Pattern. We therefore determine all method calls issued from inside methods of class c to some method u defined in the potential listener class p. The methods of class c containing the calls to p.u are considered as potential notify methods and the method p.u as update method. To test whether method c.n might be a notify method we use the predicate

isNotifyListener(c.n, p.u)**:** returns true iff c.n calls p.u and p is not a parameter of c.n.

The result of the iteration is a set Y of tuples:

$$(\mathsf{S.addListener} \mid \mathsf{removeListener}, \mathsf{S.notify}, \mathsf{L.update}).$$

To compute the final set of candidates, we iterate over the tuples of set Y. We combine corresponding addListener and removeListener methods into one pattern candidate. If the add | removeListerner entry of Y satisfies the addListener predicate defined below, we combine it with all other tuples of Y that have the same notify and update entries to associate it with the corresponding removeListener candidates. We also consider

the case that a removeListener method need not be implemented and thus always construct tuples with the removeListener entry set to null. The addListener role is defined by the predicate

isAddListener(a)**:** tests, whether the method a potentially stores the passed argument for future use, i. e., checks whether the argument

- is used on the right hand of an assignment statement, i. e., storing the argument locally in the object,
- or is passed as an argument to another method, i. e., potential call of a store method.

Figure 11 shows the static analysis algorithm we obtain.

```
Candidates := ∅

for each class c : {
  Y := ∅
  for each method m in c :
    for each parameter type p in m where (p ≯ c) :
      for each call from c.n to p.u, n and u methods :
        if (isNotifyListener(c.n, p.u))
          Y := Y ∪ {(c.m, c.n, p.u)}

  for each (c.a1, c.n1, p1.u1) ∈ Y :
    for each (c.a2, c.n2, p2.u2) ∈ Y where
        (c.n1 = c.n2 ∧ p1.u1 = p2.u2) :
      if (isAddListener(c.a1)) {
        if (c.a1 = c.a2) {
          Candidates := Candidates ∪ (c.a1, null, c.n1, p1.u1)
        } else {
          Candidates := Candidates ∪ (c.a1, c.a2, c.n1, p1.u1)
        }
      }
}
```

Figure 11: Pseudo code sketching the static search for Observer Patterns

Although the candidate set is computed quite efficiently by the directed search algorithm, we still face the problem of being too conservative with our approximation: the candidate set is large compared to

the set of actual pattern instances and not appropriate for providing it to the system designer as it is. There are three possible solutions:

Use Expert Knowledge Many approaches require expert knowledge to further restrict the candidate set. It often refers to naming conventions of methods and classes. Such approaches rely on coding discipline, which is hardly a realistic assumption in legacy codes.

In our example, we could try to eliminate methods without prefix add from the addListener candidates. However, this would also exclude register methods.

Dynamic Analyses execute the program and check if the sequence of values of variables or contents of containers is appropriate, i. e., matches the dynamic pattern. This is the approach we pursue in Section 6.2.

For the Observer Pattern, we check if the addListener method in a candidate tuple actually registers the object the notify method in the same tuple is called on.

Data Flow Analyses try to statically approximate the sequence of values that some variables have at runtime. Actually, we could formalize all rules for the dynamic matches as data flow problems. Unfortunately, data flow equations cannot be computed by a straight forward search. Instead, they require a fix point iteration and are therefore much more expensive than the simple search. Moreover, they can only make very conservative and thus mostly worthless assumptions on data provided by the user at run time. They are also imprecise in approximating object ids and aliases.

For our example, we need an alias analysis checking whether the parameter of the addListener method in a candidate tuple is an alias for the access path to the object the corresponding notify method calls update on.

6.2 Dynamic Analysis

The static analysis provided tuples of AST nodes as candidates. The dynamic analysis takes this Candidates set as its input. It monitors the execution of the nodes of every tuple. It further tracks the effects of the executed nodes to check whether the candidate satisfies the *dynamic pattern*. The dynamic pattern is a protocol (formal language) over a set of *events*. Events are state transitions of the system to analyze, e. g., assignments or method calls. In case of a protocol violation, the candidate is marked and an error message is attached to it.

Each node of a candidate tuple is contained in a class definition or is a class definition itself. At runtime we might have many instance objects of these classes. Each set of those object instances should

conform to the dynamic pattern. In our scenario, e. g., we might have more than one instantiation of the Observer Pattern defined by the subject and listener classes of a candidate tuple.

Moreover, patterns indicate $n : m$, $1 : n$, or $1 : 1$ relations among objects of the classes implementing a pattern. For each single candidate tuple, it could be required that the number of instance objects of their classes is restricted. The Observer Pattern, e. g., requires a $1 : n$ relation of the subject instances and their listener instances.

Altogether, we trace a set of instances for each candidate tuple of a pattern. Each such set may contain several objects per position in the tuple. Considering our Observer Pattern scenario, we thus assign to every candidate tuple

$$(\mathsf{S.addListener}, \mathsf{S.removeListener}, \mathsf{S.notify}, \mathsf{L.update}),$$

cf. Section 6.1, a set of instance tuples

$$\begin{aligned}\{(\quad & s.\mathsf{addListener}, s.\mathsf{removeListener}, s.\mathsf{notify},\\ & \{l_1.\mathsf{update} \ldots l_n.\mathsf{update}\} \quad)\}\end{aligned}$$

where s is an instance of S and $l_1 \ldots l_n$ are instances of L. It is not necessary to store the subject s three times. Furthermore, the addListener, removeListener, notify, and update methods are already captured by the candidate tuple. So, to avoid redundancies, we only associate a set

$$\{(s, \{l_1 \ldots l_n\})\}$$

with each candidate tuple.

We monitor each node in a tuple of the candidates. Whenever we dynamically execute such a monitored node, we retrieve all the candidate tuples the node is contained in. Depending on the node's unique role in each single tuple, we execute dynamic test actions on the object sets associated to the corresponding candidate tuples.

In the Observer Pattern, we use the subject object as a key to retrieve the affected object set of each candidate tuple. To determine the proper object set, we distinguish two cases: If the method complies with the addListener, removeListener or notify roles, then the key subject object is the object the method is called on. If the method complies with the update role, then the key subject object is the object the corresponding notify method is called on.

The dynamic test actions for the Observer Pattern are:

addListener**:** We add the passed argument to the subject's list of listener objects. No protocol mismatch can be detected here.

removeListener**:** We remove the passed argument from the subject's list of listener objects. A protocol mismatch occurs, if the listener to be removed has not been added before. This can also be caused by a programming error. We therefore allow to turn off this criterion.

notify**:** We do not change the set of subject or listener objects. A correct protocol updates all or no listener objects (atomic update). To check this protocol, we have to distinguish between the method entry and the method exit. At the method entry, we mark all attached listener objects as not-updated. At the method exit we check whether all or no listener objects have been marked as updated. In this case, the protocol is satisfied. The other case indicates a protocol violation.

To accept the case of not updating any listener objects as a protocol match makes sense, because notify may be called, although the subject's state did not change. Then there is no need to notify the attached listeners.

update**:** We do not change the set of subject or listener objects. If the update method has been called by the notify method of the same candidate tuple, we mark the listener object as updated. To recognize this, we need to detect the source of the method call, a functionality to be provided by the dynamic framework.

A call of update by the corresponding notify method is a protocol mismatch if the listener object has not been attached previously.

The dynamic analysis partitions the candidate tuples into the following categories:

Full match: Tuples contained in this category completely confirm to the dynamic pattern (protocol).

Listener objects are added via the addListener method, optionally removed with the removeListener method and their update method is called by the notify method at least one time. We distinguish $1 : n$ and $1 : 1$ matches; the latter conform to the protocol but only one listener is detected.

May match: At least one of the tuples nodes is executed, but only a correct prefix of the protocol is detected (could be completed to a correct protocol).

E. g., listener objects are added via the addListener method, but no update method is called.

Mismatch: Tuple violated the protocol requirements. The violation is logged via an error message.

No decision: None of the (monitored) nodes of a tuple is executed. Note that this category remains empty if we use a test environment, which guarantees the execution of each single program part.

7 EVALUATION

To survey our tool, we apply it to the code of the tool itself (including the Recoder package). Statistics about the tool are given in Table 2.

	classes	methods	observers[1)]
Recoder	555	6734	2
Analyzer	43	214	3
TOTAL	598	6948	5

Table 2: Statistics about the surveyed system

The main task of the static algorithm is to reduce the amount of candidates. In case of the Observer Pattern detection, it therefore applies the isAddListener and isNotifyListeners predicates as main criteria. This reduces the set of $9.7*10^{13}$ possible candidates[2)] to 28030 tuples containing all 5 Observer Pattern instances. The corresponding analysis phase needs about 70 seconds on a Pentium III, 500Mhz, 256MB RAM, running Windows NT 4 with JDK 1.3.

Table 3 shows the results of the dynamic analysis. The "Detected" row lists the numbers of tuples of the corresponding category detected by our tool, whereas the "Real" row lists the number of tuples of the corresponding category that represent real Observer occurrences.

The **Full** $1:n$ **match** category shows that all Observer Pattern instances used in that program run were classified correctly. The **Full** $1:1$ **match** column reveals that delegation confuses our analyses. The reason is, that delegation shows the same static and dynamic properties as the Observer Pattern. The only difference is that delegation always constitutes a $1:1$ relation. This is one of the reasons the static algorithm produces a lot of false positives. The following code illustrates this effect:

[1)] No subjects using delegation nor sub-classes of subjects.

[2)] It is $\binom{6948}{4} + \binom{6948}{3}$. The former term accounts for the 6948 methods in the 4 possible roles, the latter term models tuples with empty removeListener role.

	Full 1:n match	Full 1:1 match	May match	No decision	Mismatch
Detected	4	5	67	18638	9316
Real	4	0	0	1	0

Table 3: Results

```
class Delegates {
   X delegate;
   // will be detected as addListener
   void set( X x ) { delegate = x; }
   // will be detected as notify
   void internalAction() { delegate.provideFunctionality(); }
}
```

A 1 : 1 relation is suspicious, but need not be a mismatch, since this may be a valid configuration of the Observer Pattern. In case the set method of Delegates objects is called multiple times followed by a call of internalAction, our algorithm detects internalAction's violations of the notify role. In our case, all 6 tuples in the **Full** 1 : 1 **match** class were actually delegations.

The **May match** class contains no Observer instance. In over 70% of the tuples, either only the addListener or only the notify method was called, but these methods cannot provoke a protocol mismatch.

The Observer Pattern instance in category **No decision** has not been executed and, therefore, not classified. If we ensured by employing testing technology that every candidate method gets executed, we could classify all tuples and thus achieve an empty **No decision** set.

All detected mismatches were correct, i. e., these tuples did not represent an implementation of the Observer Pattern.

8 RELATED WORK

Many approaches and tools support the understanding of software structures. Some of them provide graphical views on the structure. Together from (TogetherSoft 2000) additionally supports round-trip engineering for Java and C++. This development suite performs a static analysis of the source code and visualizes the structural information as UML diagrams using an automatic graph layout.

The Pat system (Prechelt and Krämer 1998) detects structural design patterns by extracting design information from C++ header files and storing them as Prolog facts. Patterns are expressed as rules and searching is done by executing Prolog queries.

The Goose system (Ciupke 1999) gives a graphic visualization of C++ program structures using a similar approach for their detection. It uses aggregations for its static visualizations to compress the displayed information in a similar way as we do. Additionally, it detects patterns indicating design problems.

Other approaches go further in their abstractions and compute software metrics on the structure. A good overview give (H. Br et al. 1999) and (S. R. Chidamber and C. F. Kemerer 1994). (S. Demeyer, S. Ducasse, and M. Lanza 1999) define an architecture to combine software metrics and visualizations.

Online debugging and profiling techniques are state of the art. Visual debuggers support the understanding of program behavior by graph structures. GraphTrace (M. F. Kleyn and P. C. Gingrich 1988) computes static and dynamic views on an object-oriented LISP derivate. However, the static information is obtained by reflection. Hence, the tool visualizes only those program parts that are actually executed. Further static information relies on user annotations. Their architecture requires interpreted languages.

The approach of (W. De Pauw 1993) also graphically visualizes dynamic information. They use an object-to-class aggregation. The same authors developed (Jinsight 2000). It instruments a Java VM to access dynamic information about Java programs.

Scene (K. Koskimies and H. Mssenbck 1996) computes UML scenario diagrams from Oberon program executions. Their aggregation goes beyond ours: they collapse call sequences to one representative node. Their current work deals also with Java programs.

The work of (T. Ball and S. Erick 1996) combines static and dynamic information. It focuses on very large systems. It abstracts using metrics and statistics. For the visualization, it pictures scalable, colored charts instead of graphs. The tools operate post mortem; they can detect hot spots but cannot focus on them interactively or change views on the fly.

Other approaches to detect patterns mostly restrict themselves to static analyses using rather strong static signatures. These approaches fail to detect behavioral patterns as their static patterns are not distinctive enough, but their static analyses are nevertheless worth noting. We discuss some of these below.

(Keller *et al.* 1999) present static analyses to discover design patterns (Template Method, Factory Method and Bridge) from C++ systems. They identify the necessity for human insight into the problem domain of the software at hand, at least for detecting the Bridge pattern due to the large number of false positives.

(Brown 1997) additionally uses dynamic information, analyzing the flow of messages. His approach is restricted to detecting design patterns in Smalltalk, since he only regards flows in VisualWorks for Smalltalk. He therefore annotates the Smalltalk runtime environment. Another drawback is, that his approach gathers type information only at periodic events.

(Carriere *et al.* 1999) also employ code instrumentation to extract dynamic information to analyze and transform architectures. The presented approach only identifies communication primitives, but no complex protocols.

9 CONCLUSIONS AND FUTURE WORK

We presented an approach to support the understanding of software. It merges results from static program analysis with dynamic information about program execution. This combined information is presented visually with graph structures. We argued that neither static nor dynamic views by themselves provide an adequate understanding of software systems.

Large software systems require a reduction of information displayed at a time. We discussed filtering and aggregation techniques. The former reduces the data computed in the static analysis the latter reduces the information in a view while the data is still present in the view model. Both techniques are applied in our approach.

We instantiated two frameworks, one for analyzing static information from program representations (Recoder), the other for extracting dynamic information from program executions and for the visualization of information (VizzEditor). The resulting tool, the VizzAnalyzer, implements the approach of merging static and dynamic views on programs. It supports the analysis of design flaws and supports the analysis phase of the reengineering process in software systems.

In the future, the architecture could be used as a visual tool to also control source code modifications. Recoder already provides structures and methods for source code transformations (meta programming). Often, the effects of such operations are hard to understand especially if they are done at run time (dynamic meta programming). The VizzAnalyzer could visualize the transformations as well as the new program behavior. Additionally, user defined aggregation could be used to trigger a restructuring of packages or subclass hierarchies.

Another direction of future work is the integration of metric or other more abstract views into the architecture. The integration of statistic profiling views would open new fields of applications like program optimizations.

The Recoder software is developed in an open source project. It can be downloaded from:

`http://recoder.sf.net`

The VizzEditor and VizzAnalyzer home page is:

`http://i44pc29.info.uni-karlsruhe.de/VizzWeb`

A more elaborated presentation of the VizzAnalyzer approach and tool gives (A. Schwind 2000).

The present paper shows how to detect communication patterns in legacy systems. Therefore, we filtered information gained by static analysis using the results of dynamic analysis. This approach improves the quality of the results tremendously as protocol conformance of a pattern can be checked. Moreover, we partitioned the candidate pattern instances into the categories **Full match**, **May match**, and **Mismatch**

giving more differentiated information to the user.

One drawback of our current implementation concerns the dynamic analyses: If a pattern candidate instance is not executed during a concrete program run, our dynamic analyses cannot provide any evidence for its conformance to or its violation of the pattern rules. We will avoid this problem using results from testing theory. These ensure that every point of a program gets executed at least once. To further improve our static analyses, we plan to integrate data flow analyses and checking of naming conventions.

Another deficiency of our currently implemented static analysis concerns Observer Patterns implemented by delegation: If a subject class provides its functionality by delegating calls to addListener, removeListener and notify to the corresponding methods of another subject class, our static analysis currently does not recognize both classes as subject candidates. To deal with this, we are currently implementing the following algorithm using our Recoder tool: Calculate all calls to the addListener, removeListener and notify methods of a class already considered a subject candidate. If these calls origin from different methods of *one* other class, then also consider this class a potential subject.

Another direction of future work is the framework extension to support more patterns and anti-patterns (Brown *et al.* 1998), as well. Since implementing static and dynamic algorithms by hand is a costly concern, we need to develop a tool generating analysis programs from pattern specifications. We also like to leverage the benefit of our approach by simultaneously detecting multiple patterns.

Finally, we strive to improve visualization of detected patterns to increase user support for understanding large scale software systems.

REFERENCES

[Bass *et al.* 1998] Bass, L., P. Clement, and R. Kazman (1998), *Software Architecture in Practice*, Addison Wesley.

[Brown 1997] Brown, K. (1997), "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk," .

[Brown *et al.* 1998] Brown, W. J., R. C. Malveau, H. W. S. McCormick III, and T. J. Mowbray (1998), *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley, New York, NY.

[Carriere *et al.* 1999] Carriere, S. J., S. G. Woods, and R. Kazman (1999), "Software Architectural Transformation," In *Proceedings of WCRE 99*.

[Ciupke 1999] Ciupke, O. (1999), "Automatic Detection of Design Problems in Object-Oriented Reengineering," In *Technology of Object-Oriented Languages and Systems - TOOLS 30*, IEEE Computer Society, pp. 18–32.

[Gamma *et al.* 1995] Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company, New York, NY.

[Keller *et al.* 1999] Keller, R. K., R. Schauer, S. Robitaille, and P. Page (1999), "Pattern-Based Reverse-Engineering of Design Components," In *International Conference on Software Engineering*, pp. 226–235.

[Prechelt and Krämer 1998] Prechelt, L. and C. Krämer (1998), "Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns," *J.UCS: Journal of Universal Computer Science 4*, 12, 866ff.

[Shaw and Graham 1996] Shaw, M. and D. Graham (1996), *Software Architecture in Practice – Perspectives on an Emerging Discipline*, Prentice Hall.