Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information

Tamar Richner and Stéphane Ducasse
Software Composition Group, Institut für Informatik (IAM)
Universität Bern, Neubrückstrasse 10, 3012 Berne, Switzerland
{richner,ducasse}@iam.unibe.ch
http://www.iam.unibe.ch/∼{richner,ducasse}

to appear in Proceedings ICSM'99, IEEE

Abstract

Recovering architectural documentation from code is crucial to maintaining and reengineering software systems. Reverse engineering and program understanding approaches are often limited by the fact that (1) they propose a fixed set of predefined views and (2) they consider either purely static or purely dynamic views of the application. In this paper we present an environment supporting the generation of tailorable views of object-oriented systems from both static and dynamic information. Our approach is based on the combination of user-defined queries which allow an engineer to create high-level abstractions and to produce views using these abstractions.

Keywords: *architectural recovery, program understanding, object-oriented reverse engineering, dynamic analysis*

1. Introduction

Understanding the structure and behavior of an application being developed or maintained is essential throughout the software development cycle. While documentation should address this need, it is often neither complete nor up-to-date and may not at all address the particular questions an engineer is interested in. Recovering such information from an existing application is an important aid to engineers confronted with a variety of software engineering tasks.

In reverse engineering we seek to recover information about design decisions which were taken in developing the software. Just as in forward engineering there is a recognized need for a variety of modeling techniques, or architectural views [2][20], so in design recovery we need to be able to generate a range of views mined from both structural and behavioral information about the code.

Reverse engineering and maintenance of object-oriented applications presents special challenges [33]. In contrast to procedural applications, where a specific functionality is often identified with a subsystem or module, the functionality in OO systems comes from the cooperation of interacting objects and methods. Detecting and deciphering these interactions in the source code is not easy: polymorphism makes it difficult to determine which method is actually executed at runtime, and inheritance means that each object in a running system exhibits behavior which is defined not only in its class, but also in each of its superclasses. This difficulty is further aggravated in the case of dynamically typed languages like Smalltalk where no type definition is available at compile time and where methods are never statically bound.

Tools like Sniff+ [29] and CIA++ [11], have been developed to support object-oriented application understanding based on static information only. Their use for reverse engineering remains limited, however, because of the difficulty of filtering relevant from irrelevant information in the great mass of data extracted. Program slicing techniques could aid in code understanding by focusing on a relevant aspect of a system. But slicing techniques which work for procedural languages do not adapt well to OO languages since dynamic binding and polymorphism make it difficult to obtain precise control flow information. Some of these techniques have recently been extended to handle features of statically typed OO languages [22][31].

The use of dynamic information is an attractive option in the reverse engineering of object-oriented software for several reasons: first, an executed scenario is like a program slice – it limits our scope of investigation. Second, dynamic information is always precise with respect to the executed scenario. Thirdly, obtaining an execution trace is relatively simple as compared with control flow analysis required for obtaining a program slice statically. Finally, an execution trace provides some information which can never be extracted from static analysis, such as the number of instances or the multiplicity of relationships between objects.

The main argument against the use of dynamic information is its incomplete coverage of the code. In the context of reverse engineering and program understanding this is not necessarily a disadvantage. For program understanding we do not always need complete information: we need information that helps us to form concepts about the software structure and helps us to formulate new hypotheses and questions.

In this paper we show how we use a logic programming language to query both static and dynamic information and to generate a range of high-level views for object-oriented applications. Static information about the program is represented by a meta-model which reifies object-oriented notions such as class, superclass, inheritance and attribute. This structural information is complemented with behavioral information in the form of traces from program execution.

Our approach is related to work in reverse engineering and in the use of dynamic information for program understanding. Reverse engineering techniques traditionally make use of static information only, restricting the kinds of questions which can be answered. On the other hand, program understanding approaches which use dynamic information present very fine-grained views of an application. Such approaches are confronted with the challenge of extracting high-level views from a program trace, and have so far focused mostly on visualization techniques for the large amount of information generated through program execution [13].

The contribution of our approach is to allow an engineer to steer his or her investigation of the code through an iterative process similar to that described by Murphy [24]. This is possible because the engineer can specify declaratively the kinds of views which are of interest. An initial view answers some questions and introduces new ones, and views can be refined to different levels of abstraction. Since dynamic information is available, as well as static, a large range of questions can be answered.

To evaluate our approach we have applied it to several Smalltalk applications – one case study is presented in the paper. It demonstrated the flexibility of the approach, and encouraged us to continue our work in refining the analysis of dynamic information and the generation of views.

The rest of the paper is then structured as follows: In section 2 we show how we model object-oriented programs and their execution using a logic programming language. In section 3 we describe how high-level views of an application can be created based on the model presented in section 2. Section 4 presents a case study to illustrate our approach. Section 5 briefly discusses the implementation, and section 6 presents related work. Finally, section 7 concludes with a discussion, evaluating our contribution and pointing to direction for future work.

2. Modeling OO Programs and Their Execution

We model static and dynamic aspects of an object-oriented application in terms of logic facts. Both static and dynamic information are stored in a single logic database. This section presents a model for static and dynamic execution information which allows us to form queries that combine both aspects.

2.1. Basic Relations

Static Relations. We represent static information about an application using Prolog facts (see table 1). This information is extracted using static analysis tools and represented in the FAMIX model (see section 5 for more details).

Table 1. The basic static relations

class(ClassName, SourceAnchor) a class and its source artifact
superclass(SuperClass, SubClass) an inheritance relationship
attribute(Class, AttributeName, AttributeType) class defines an attribute of a certain type
method(Class, MethodName, IsClassMethod, Category) a class defines a method belonging to a category
access(Class2, Attribute, Class1, Method) an attribute of Class1 is accessed by Method of Class2
invocation(Sender, Method, ReceivedMethod, Candidates) Method of Sender invokes ReceivedMethod on one of the Candidates

Note that the parameter Candidates of predicate invocation gives the potential receivers of the invocation; an empty list means that no candidates were found within the classes of the application.

Below is a sample which provides information about the class EllipseFigure of the HotDraw framework [1].

```
(1) class('EllipseFigure','HotDraw-Figures').
(2) superclass('Figure','EllipseFigure').
(3) method('EllipseFigure','displayFilledOn:',false,'displaying').
(4) access('EllipseFigure','self','EllipseFigure','displayFilledOn:').
(5) invocation('EllipseFigure','displayFilledOn:','fillColor',['Figure']).
(6) invocation('EllipseFigure','displayFilledOn:','paint:',[]).
```

Dynamic Execution. Dynamic information is represented as facts about method invocations in a program's execution (table 2). These are numbered according to sequence order (SN) and stack level (SL). Each send or indirectsend fact corresponds to to the invocation of an observed method on an instance of a class. As an example, the send facts listed below record the method invocations that follow the invocation of EllipseFigure(instance#39).fillColor. This sequence

Table 2. The basic dynamic relations

send(SN, SL, Class1, I1, Class2, I2, M2) an instance I1 of Class1 invokes method M2 on instance I2 of Class2. SN is the sequence number of the event, and SL is the stack level of the method call.
indirectsend(SN, SL, Class1, I1, M1, Class2, I2, M2) an instance I1 of Class1 sends the message M1, which is unobserved. The next observed invocation is the execution of method M2 on instance I2 of Class2.

corresponds to one execution of the invocation described in line 5 of the static information above.

```
send(5,7,'EllipseFigure',39,'EllipseFigure',39,'fillColor').
send(6,8,'EllipseFigure',39,'Drawing',85,'fillColor').
send(7,9,'Drawing',85,'FigureAttributes',26,'fillColor').
send(8,9,'Drawing',85,'FigureAttributes',26,'fillColor').
send(9,7,'EllipseFigure',39,'Drawing',85,'compositionBoundsFor:').
```

2.2. Derived Relations

The basic relations represent facts about the source code and its execution. This section presents rules which form a logic layer above the database of facts and allow more sophisticated queries about the structure and behavior of the application.

Querying static information. Rule 1 below specifies that a subclass Subclass overrides a method Method defined in a class Class. It makes use of rules 2, commonMethod, which says that Class1 and Class2 define a method with the same name Method, and rule 3, inHierarchy, which defines what it means for a class to be in the inheritance hierarchy of another class.

```
rule1:  overrides(Class, Subclass, Method) :-
            commonMethod(Class1, Class2, Method),
            inHierarchy(Class, Subclass).

rule2:  commonMethod(Class1, Class2, Method) :-
            method(Class1, Method, IsClassMethod, _),
            method(Class2, Method, IsClassMethod, _).

rule3:  inHierarchy(Class, Class).
        inHierarchy(Class,Subclass) :-
            superclass(Class,Subclass).
        inHierarchy(Class,Subclass) :-
            superclass(Superclass,Subclass),
            inHierarchy(Class,Superclass).
```

Querying dynamic information. Rule 4 below specifies that an instance of Class1 invokes, either directly or indirectly, a Method on an instance of another class, Class2.

```
rule4:  invokesMethodClass(Class1,Class2,Method) :-
            send(_,_,Class1,_,Class2,_,Method),
            not(equal(Class1,Class2)).

        invokesMethodClass(Class1,Class2,Method) :-
            indirectsend(_,_,Class1,_,_,Class2,_,Method),
            not(equal(Class1,Class2)).
```

Queries combining static and dynamic information. Rule 5 below specifies that an instance of Class1 invokes a Method on an instance of a metaclass of Class2, where the Smalltalk category of Method is instance creation (Smalltalk creation methods, defined at the class level, appear as methods of a metaclass in our model. There is an implicit Smalltalk convention to group instance creation methods into a category named instance creation). Rule 6 allows one to go up the inheritance hierarchy to find the Smalltalk category of the Method, in case it is not defined by Metaclass. Rule 5 thus defines a *create* relationship between two classes – an instance of class Class1 creates an instance of Class2 – and will be used in the case study to generate a *creation* view (see section 4).

```
rule5:  sendsCreate(Class1,Class2) :-
            invokesMethodClass(Class1,MetaClass,Method),
            metaclassOf(MetaClass,Class2),
            methodCategory(MetaClass,Method,'instance creation').

rule6:  methodCategory(Class,Method,Category) :-
            method(Class,Method,_,Category).

        methodCategory(Class,Method,Category) :-
            inHierarchy(Superclass,Class),
            method(Superclass,Method,_,Category).
```

Discussion. Though the model we present for representing OO programs and their execution is language-independent, its interpretation is not. For example, in table 1, the interpretation of SourceAnchor in the relation class(ClassName,SourceAnchor) is language-specific. In Smalltalk this will correspond to a class category, in C++ to a file name and in Java to a package name. Furthermore, whereas in Smalltalk no type information is available for attributes, or for precise identification of receiver candidates in an invocation, this kind of information can be easily obtained from statically typed languages like C++ and Java. Thus, some of the rules for the derived relations hold for all OO languages, for example overrides(Class,Subclass,Method), whereas others, such as sendsCreate(Class1,Class2) are more language specific.

A querying approach similar to ours, but using static information only, has also been applied by [18][27][36] where rules are used for the detection of certain structures or their violations. In Program Explorer [21] both static and dy-

namic information are represented as logic facts. The static information is used to filter out execution events to be visualized, in order to support the discovery of design pattern in C++ code.

Our goal, however, is not only to detect certain structures. Our approach supports the specification and generation of a wide range of views of an application to aid in architectural recovery. The next section discusses the specification and generation of views.

3. Generating High-Level Views

We define a view of an application as a set of components and the connectors between them [28]. In a view two components C_1 and C_2 are connected by a connector of type R if there exists at least one member e_1 of C_1 and one member e_2 of C_2 such that $R(e_1,e_2)$ holds. An element e is a member of at most one component C. To specify a view we then use Prolog rules to define:

(1) a clustering to components $C_1, C_2, ..., C_N$, which defines a partition on $E = \{e_1, e_2, ...e_m\}$, the set of all the elements e in our model.

(2) a relation $R : ExE \rightarrow \{0, 1\}$, which specifies whether or not a certain relationship holds between two elements.

For example, to generate a view that shows the message sends (from an executed scenario) between the Smalltalk class categories of the HotDraw framework we define a relation R and a clustering C using rule 7 and 8 respectively, as shown below. In this case E is the set of all classes in the HotDraw framework, and is defined implicitly by rule 7. The Prolog query createView(invokesClass,allInCategory) then generates a view which is displayed as a graph (using the dot tool [17]), as seen in Figure 1. Each node in the graph corresponds to a HotDraw class category and each directed edge $A \rightarrow B$ means that at least one instance of a class in category A invokes a method on an instance of a class in category B.

```
createView(invokesClass,allInCategory).

R rule7:  invokesClass(Class1,Class2) :-
              invokesMethodClass(Class1,Class2,_).

C rule8:  allInCategory(Category,ListOfClasses) :-
              setof(Class,class(Class,Category),ListOfClasses).
```

View 1: Dynamic invocations between class categories.

This view gives us a coarse idea of the communication between parts of the HotDraw framework. However, since the category HotDraw-Framework groups together some of the main classes, this clustering must be broken down to get a better understanding of the behavior of the application. This will be elaborated on in section 4.

Our prototype tool, Gaudi, defines basic and derived relations as described in section 2, as well as functions to generate views using these relations. Using these relations

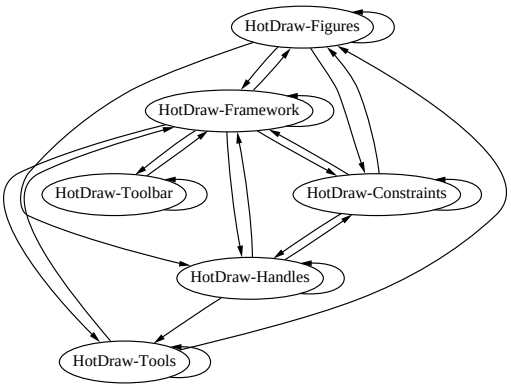

Figure 1. Invocations between class categories

a large number of relationships R between basic elements and of clusterings C are possible. Some of these will be illustrated in the case study which follows.

4. Case Study: Understanding HotDraw

In this section we demonstrate how our approach supports reverse engineering and program understanding by applying it on the HotDraw framework [1]. We show how Prolog rules are used to define R and C for specifying and generating a range of views of the framework. We first briefly present HotDraw. We then define a scenario and specify and analyze some views generated from this trace information.

4.1. The HotDraw Framework

HotDraw is a framework for semantic graphic editors. It allows for the creation of graphical editors which associate the picture with a data structure - that is, changing the picture also changes the data structure. The HotDraw framework consists of 114 Smalltalk classes and comes with several sample editors.

From the documentation we learn that HotDraw is based on the Model-View-Controller triad [19]; these roles are played by the classes Drawing, DrawingEditor and DrawingController respectively. Furthermore, it has a few other basic elements: *tools* are used to manipulate the drawing which consists of *figures* accessed through *handles*. *Constraints* are used to ensure that certain invariants are met, for example, that two figures connected with a line remain connected if one of the figures is moved.

Though HotDraw has been documented in several publications, including patterns for customizing the framework [14], the overall view of the framework is never described. Moreover, several changes have been made to the framework over the years, in particular to the implementation of tools and of constraints, and most of the documentation is out of date with respect to the version 3.0 we are using.

4.2. A Scenario

To fill the logic database we parsed the code to obtain a static model of HotDraw [30]. To obtain the dynamic information we instrumented all the methods defined for classes in the HotDraw categories using Method Wrappers[3], and ran a typical scenario on one of the sample applications, DrawingEditor, to generate an execution trace.

The scenario we ran consisted of the creation of several kinds of figures (rectangle, rounded rectangle, bezier, text and image), deletion of a figure, grouping and ungrouping of two figures (rectangle and rounded rectangle), cutting and pasting one of the figures, moving a simple figure and a grouped figure, changing fill and line color of some of the figures and finally quitting the application.

4.3. Reverse Engineering HotDraw

1. Refining the high-level model. Our starting point is the initial view generated, as shown in Figure 1. This view shows all traced communication between instances, as grouped by the Smalltalk category to which the class belongs.

Analysis. This view gives us some rough idea of the relationships of the main parts of HotDraw. We see that that the HotDraw-Toolbar component communicates only with the HotDraw-Framework component, that HotDraw-Constraints communicates with HotDraw-Figures and HotDraw-Handles as would be expected.

We make, however, two observations with respect to this initial view. First, since the HotDraw-Framework category contains several of the main HotDraw classes, in particular DrawingEditor, Drawing and DrawingController, we want to view these separately. Second, we do not necessarily want to see all the invocations in one view. In particular, we want to distinguish between invocations which create instances, and those in which instances just invoke methods on each other.

2. Clustering. We therefore first define a new component breakdown as shown below. This clustering creates five components: Figure, Handle, Constraint, Toolbar and Tool. Classes which are not mapped into components are considered themselves components. This clustering is used to view the creation invocations and the simple invocations (see Figure 2 and Figure 3).

```
component('Figure',L]) :- allInCategory('HotDraw-Figures',L).
component('Handle',L) :- allInCategory('HotDraw-Handles',L).
component('Constraint',L) :- allInCategory('HotDraw-Constraints',L).
component('Toolbar',L) :- allInCategory('HotDraw-Toolbar',L).
component('Tool',L) :- allInCategory('HotDraw-Tools',L).
```

3. Creation Invocations. To distinguish between invocations which create instances and other kinds of invocations, we define a rule which specifies a *create* relationship between classes.

```
rule9: sendsCreate(Class1,Class2) :-
           invokesMethodClass(Class1,MetaClass,Method),
           metaclassOf(MetaClass,Class2),
           methodCategory(MetaClass,Method,'instance creation').

       sendsCreate(Class1,Class2) :-
           indirectsend(_, _,Class1, _,'new',Class2, _, _).
```

The first part of rule 9 – same as rule 5 in section 2.2 – captures the events in which a method belonging to the instance creation Smalltalk method category is invoked on a metaclass. The second part of rule 9 captures the events in which Class1 sends an unobserved (not instrumented) method new, resulting in an invocation of some method on Class2. We now use the clustering defined above to create *creation* view in which these components appear, as shown in Figure 2.

```
createView(sendsCreate,component).

R  sendsCreate(Class1,Class2).

C  component(ComponentName,ListOfClasses).
```

View 2: Creation invocations between components.

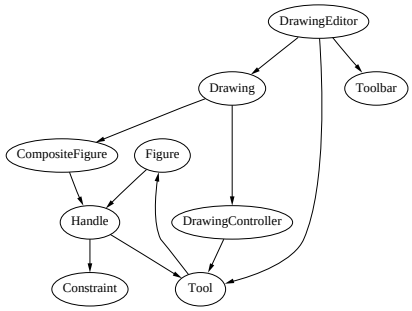

Figure 2. Creation invocations between components

Analysis. This view (Figure 2) tells us that DrawingEditor creates Drawing, which creates DrawingController, as expected from the Model-View-Controller pattern. In contrast to the other figures which are created by Tool, CompositeFigure is created by Drawing. As the components Handle, DrawingEditor and DrawingController all point to Tool, there is some ambiguity about the creation of this component. To find more about the creation responsibilities there we must split the Tool component into its constituent parts to get a closer look. We first, however, create a view of non-creation invocations between the components.

4. Non-creation Invocations. We want to obtain a high-level view of the simple calls between instances, as shown

in Figure 3. To do this we define a rule sendsSimple(Class1,Class2) which gives us all the invocations between instances which do not correspond to creation invocations. (Note: for the sake of brevity, we do not give the definitions of the Prolog rules from here on). We also exclude all invocations between an instance of a metaclass A and an instance of a class A, making the assumption that these invocations are probably associated with creation events.

createView(sendsSimple,component).	
R	sendsSimple(Class1,Class2).
C	component(ComponentName,ListOfClasses).

View 3: Invocations between components.

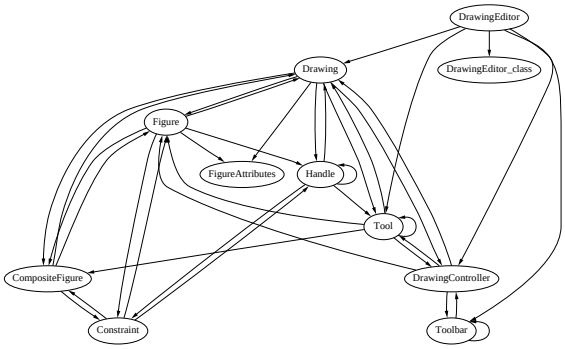

Figure 3. Invocations between components

Analysis. From Figure 3 we see that Drawing is central to the application – it communicates with Figure, Tool, Handle, DrawingEditor and DrawingController. FigureAttributes is invoked only by Drawing and Figure, and as the name suggests, probably stores attributes like line width and color which are relevant for all figures in the drawing. Constraints, figures and handles form a triad: this makes sense since we know already from the creation view that figures create handles which in turn create constraints. The Toolbar component is relatively independent of the other components and invokes only methods on DrawingController.

5. About Tools. We want to get a better understanding of how tools are implemented. For this we generate a new view which gives us the creation relationships between the classes, rather than the components. Furthermore, we want to know the multiplicity of these relationships. For example, whether each instance of a figure create only one or several instances of a tool. The view in Figure 4 below therefore displays two kinds of edges. A filled edge $A \rightarrow B$ means that there is an instance of class A which creates only one instance of class B. A dashed edge $A \rightarrow B$ means that there is an instance of Class A which creates several instances of class B.

Analysis. From this view (Figure 4) we see that instances of the class Tool are created by DrawingEditor, and that one

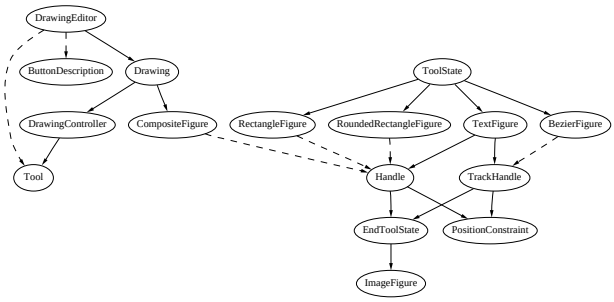

Figure 4. Multiplicity of creation invocations

DrawingEditor creates several instances. The class DrawingController also seems to create one instance of Tool; looking more closely at this, through querying about this creation event, we discover that when the application starts up and DrawingController is initialized, it invokes selectionTool on an instance of Tool_class. The other tools are created later when the DrawingEditor creates a tool for each icon in the toolbar.

In Figure 4 we had expected to see a creation relationship between Tool and ToolState. To understand this singularity we browsed the code and found that the Tool class (Tool_class in our model) instantiates ToolState at load-time using class methods, and so instances of ToolState were created before the instrumented application was run. We also discovered by browsing that Tool class also creates an instance of EndToolState representing the last state of the tool state machine. Going back to the view in Figure 4 we wondered why instances of Handle and TrackHandle were creating more instances of EndToolState. Again, by browsing the code, we understood that a Handle contains a per default action that is to return to the end state of the tool state machine.

6. Tool, ToolState, SimpleTransitionTable. We now have an inkling about the implementation of tools using state machines. To understand this in greater depth we create a view which shows the communication between instances. By detecting the invocation sendCreate(ToolState,RectangleFigure) we identify a trace sequence corresponding to a mini-scenario around the creation of an instance of RectangleFigure. We then display the invocations between instances (Figure 5) in a form similar to a UML collaboration diagram [2]: invocations are numbered sequentially in the order they occur; missing numbers correspond to self sends - these have been omitted in order to make the graph more comprehensible. The one dashed edge corresponds to a creation event.

Since the methods being invoked do not appear in this view[1], we generate a view which clusters method of a class

[1] Our initial view labeled the edges with the name of the method being invoked, but the resulting graph is difficult to display here.

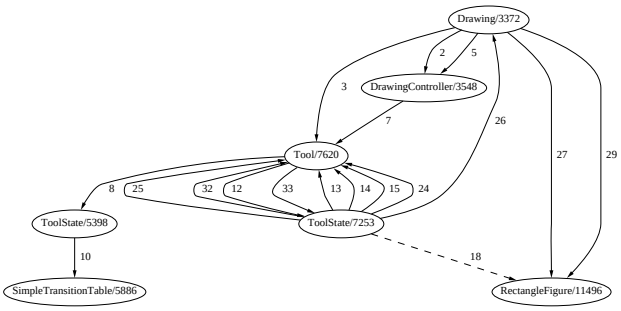

Figure 5. Instance invocations around creation of a Rectangle figure

A which are called by class B in our mini-scenario. The resulting view is displayed in Figure 6. It is to be interpreted as follows: on the instance of Tool, the instance of Drawing invokes the method passInputDown, the instance of DrawingController invokes handleEvent and so on.

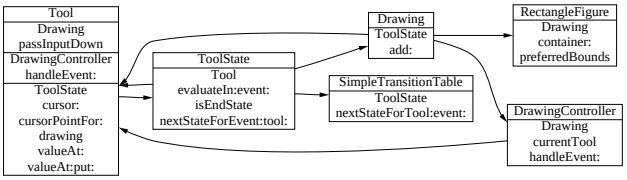

Figure 6. Method groupings according to invoking class

Analysis. With the help of figure 6 and by browsing the code, we interpret this collaboration: when a user event happens Drawing invokes DrawingController.currentTool to get the identity of the appropriate tool. It then passes input down to the Tool and tells the DrawingController to handle the event. The DrawingController then tells the Tool to handle the event. The Tool consults with the current tool state ToolState/5398 (which in turn consults the SimpleTransitionTable) to get the next tool state for the event. It then tells the appropriate state ToolState/7253 to handle the event. ToolState/7253 creates an instance of RectangleFigure, and tells the Drawing to add this figure to itself. Drawing then repairs itself by asking RectangleFigure about its bounds.

4.4. Lessons Learned

This experiment in reverse engineering HotDraw shows that our approach supports the exploration of object-oriented applications. The clustering of entities allowed us to obtain several views with a different granularity going from sets of classes grouped together to interaction between instances. Moreover, the combination of static and dynamic information helped to narrow our understanding following the scenario.

A view works as a catalyst for generating questions about the studied system and helps the engineer to focus the investigation of the code. It confronts the engineer with a new model to compare to his or her initial mental model and assumptions about the system. A view not only helped us to understand HotDraw by showing relationships between the entities, such as the creation between components, but also by provoking questions about the absence of expected relationships or the presence of unexpected ones.

We have presented views as a combination of a simple relationship $R(e_1, e_2)$ and a clustering C. Our prototype allows us to display also relationships in which arcs between components are labeled. We are as well investigating different kinds of views and how they are best displayed graphically, within the limitations of a simple graph layout tool.

5. Implementation

Our environment for query and view recovery has been constructed as an integration of tools for extraction, analysis and visualization.

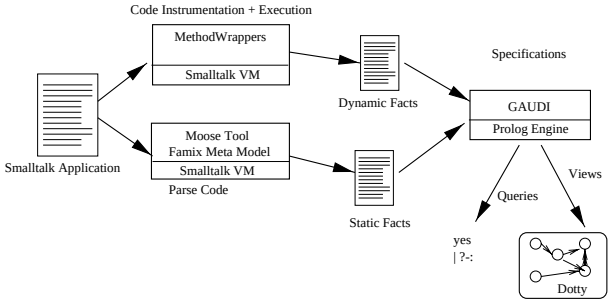

Figure 7. View recovery from Smalltalk applications

We used the MOOSE tool to parse the Smalltalk code and represent it in the FAMIX model (FAMoos[2] Information EXchange model, see [30]). This model defines a core data model for object-oriented concepts – Class, Method, Attribute and InheritanceDefinition– plus the necessary associations between them – Invocation and Access, and can be extended using meta-modeling techniques.

The Smalltalk applications were instrumented using Method Wrappers [3], which allow instrumentation at the level of individual methods. The tracing information obtained from executing an application was then written to a file as Prolog facts. Gaudi, written in SICTUS Prolog [25], consists of a set of rules defining relations and clusters used

[2]The FAMOOS ESPRIT project investigates tools and techniques for transforming object-oriented legacy systems into frameworks. See *http://www.iam.unibe.ch/∼famoos/*

to create the views, and of functions to generate views. The dot tool [17] was used to display the views generated by Gaudi.

Discussion. We have applied our approach so far on Smalltalk applications. It could, however, easily be adapted to other class based object-oriented languages like C++ and Java. Our current model does not support concurrent objects; one way to deal with concurrency would be to introduce the notion of process in the meta model and to attach the execution instances to these processes.

6. Related Work

Although some work on program analysis tools [6] and debuggers [23] is related to ours, these kinds of tools have a different goal than reverse engineering tools and are not well adapted to creating high-level models and revealing overall structures in the software. Here we relate our work to research on reverse engineering, the use of dynamic information for program understanding and declarative reasoning about program structure.

Reverse engineering. Reverse engineering approaches all seek to represent the software at a higher level than that of the information which is directly extracted from the code. They differ, however, in their solutions to the following main issues: the data model on which the tool operates, the strategy for creating a high-level model and the kinds of views offered. The MANSART tool [12], as well as the approach described in [10], requires information obtainable from an abstract syntax tree (AST) of the program, and uses 'recognizers' to detect language-specific clichés associated with specific architectural styles. Each style can then be viewed as a simple graph. Rigi [35] and the reflexion model tool [24] both use any set of relations extracted from the code. The Rigi tool incorporates automatic clustering, but also allows user defined grouping of the source model. It allows for hierarchically embedded views of different relations and presents a sophisticated user interface for manipulating these. The Reflexion model approach expects an engineer to define a high-level model and a declarative mapping from the source relations to this model. Its view then reports how close the high-level model comes to describing the source code. Dali [15] is a workbench which integrates several extraction tools and allows for the combination of the views obtained from these different sources.

In terms of the data model required, our approach operates on basic structural information about the OO code, complemented by relations from program traces. Our strategy for building a high-level model is, as in Murphy's approach [24], to leave it up to the engineer. Finally, since the engineer can formulate hypotheses declaratively over the whole static and dynamic model, this means that a large range of views can be obtained. We see the strength of our approach in the flexibility it provides for tailoring views to the questions of the engineer.

Use of dynamic information in program understanding. Program traces have been used in software maintenance to locate code implementing a particular program feature [34], to extract business rules from COBOL programs [26], and to discover program invariants [9].

For the understanding of object-oriented software, much of the work on using dynamic information has focused on techniques for visualizing the large amount of information [8][16]. Program Explorer [21] allows the filtering of events or objects of interest using static and dynamic information, but abstractions of a granularity greater than a class can not be viewed. ISViS [13] is a visualization tool which displays interaction diagrams using a mural technique and also offers pattern matching capabilities to aid in identifying recurring patterns of events. Few tools offer architectural level visualizations. Sefika et al.[27] can display the interactions of architectural units such as subsystems, but their approach requires an instrumentation specific to the application. More recently, Walker et al.[32] use program animation techniques to display the number of objects involved in the execution, and the interaction between them through user-defined high-level models.

We view our approach as complementary to techniques like ISViS [13], or Walker et al.[32]. Like these techniques, our work is based on the recognition that object level information from program traces is too fine grained for architectural understanding. However, whereas Walker et al. present an animation technique geared primarily for performance tuning, and ISViS focuses on visualizing and detecting interaction patterns, our goal is to allow an engineer to specify the kinds of views that best suit his or her investigation.

Declarative reasoning about program structure. Prolog has been used as an inference engine for other reverse engineering approaches, such as for the understanding of dataflow in Pascal programs [5] and for reasoning about dependency relationships between modules [7]. As mentioned in section 2, it has also been used to query static program information in order to find structural design patterns[18] and to detect violations of programming conventions and rules [27]. The detection of behavioral design patterns may be easier with the incorporation of dynamic information; the discovery of architectural patterns [4] presents a more challenging problem. In SOUL [36] a logic-programming language is integrated in the Smalltalk development environment, allowing one to reason about static structure and to enforce design rules and conventions.

7. Conclusions and Future Work

We have presented an approach to reverse engineering object-oriented applications based on the use of a logic programming language for recovering views of the code.

As argued in the introduction, dynamic information, though incomplete, is useful in reverse engineering because it acts like a program slice. Obtaining dynamic information, however, requires that the system be executable (and instrumentable) – and so this approach will not work for parts of systems, or other code which can not be executed.

Another question is the scalability of our approach. To reduce the amount of trace information generated, tracing can be more sparingly used, for example by instrumenting the application at only some of the methods or classes. A clever solution would be a feedback from the query results to the instrumentation so that only relevant methods are instrumented. The choice of scenario executed also plays a role in the amount of information generated. Finally, before analysis is done, the trace information could be filtered to keep only the relevant events.

The strength of our approach is the flexibility it offers in two respects: the kinds of views which can be recovered, and the kinds of questions which can be answered. First, our approach is not restricted to generating a fixed set of views, but allows an engineer to define views of interest by declaratively defining the relations to be displayed and the clustering to be applied. This makes it possible to create views of varying granularity and so to ask questions at different abstraction levels.

Second, since our approach uses both static and dynamic information, it can answer a large range of questions about an application: where static information is less focused it is complemented by dynamic information; static information is used to cluster dynamic information into more manageable components and dynamic information provides answers to questions which cannot be answered with static information only.

Finally, as seen from the case study presented, the views generated work as catalysts for generating questions about the studied system and helping to steer the process of design recovery. Much as in Reflexion models[24], this iterative process of comparing an expected model with a recovered view is at the heart of program understanding and reverse engineering.

We are continuing this work in applying our approach to several Smalltalk applications. This will allow us to refine our analysis rules and to discover which kinds of views could be most useful in reverse engineering. We are also interested in learning about the process of applying such a tool in order to be able to offer guidelines on its use in reverse engineering.

Acknowledgments We would like to thank Serge Demeyer and Oscar Nierstrasz for their comments on the paper. We also thank the anonymous reviewers for their comments and for pointing us to some related papers. This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT program Project no. 21975.

References

[1] K. Beck and R. Johnson. Patterns generate architectures. In *Proceedings ECOOP'94*, LNCS 821, pages 139–149. Springer-Verlag, July 1994.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[3] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP'98*, LNCS 1445, pages 396–417. Springer-Verlag, 1998.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stad. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley, 1996.

[5] G. Canfora, A. Cimitile, and U. de Carlini. A logic-based approach to reverse engineering tools production. *Transactions on Software Engineering*, 18(12):1053–1064, Dec. 1992.

[6] G. Canfora, A. Cimitile, U. de Carlini, and A. D. Lucia. An extensible system for source code analysis. *Transactions on Software Engineering*, 24(9):721–740, Sept. 1998.

[7] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *Proc of the International Conference on Software Engineering*, pages 138–156, 1992.

[8] W. DePauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In *Proceedings ECOOP'94*, LNCS 821, pages 163–182. Springer-Verlag, July 1994.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of ICSE'99*, May 1999.

[10] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. A cliché-based environment to support architectural reverse engineering. In *Proceedings ICSM '96*. IEEE, Nov. 1996.

[11] J. Grass. Object-oriented design archeology with CIA++. *Computing Systems*, 5(1):5–67, 1992.

[12] D. Harris, A. Yeh, and H. Reubenstein. Extracting architectural features from source code. *Automated Software Engineering*, 3(1-2):109–139, 1996.

[13] D. Jerding and S. Rugaber. Using Visualization for Architectural Localization and Extraction. In *Proceedings WCRE*, pages 56 – 65. IEEE, 1997.

[14] R. E. Johnson. Documenting frameworks using patterns. In *Proc of OOPSLA '92*, pages 63–76, Oct. 1992.

[15] R. Kazman and S. J. Carriere. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse*, Victoria, B.C., 1998.

[16] K. Koskimies and H. Mössenböck. Scene: Using scenario diagrams and active test for illustrating object-oriented programs. In *Proceedings of ICSE-18*, pages 366–375. IEEE, Mar. 1996.

[17] E. Koutsofios and S. C. North. *Drawing graphs with dot*. AT & T Bell Laboratories, Murray Hill, NJ.

[18] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedingsof WCRE '96*. IEEE, Nov. 1996.

[19] G. E. Krasner and S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *JOOP*, pages 26–49, Aug. 1988.

[20] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.

[21] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA'95*, pages 342–357. ACM Press, 1995.

[22] L. Larsen and M. Harrold. Slicing object-oriented software. In *Proceedings ICSE '96*, pages 495–505. IEEE, 1996.

[23] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proceedings OOPSLA '97*, ACM SIGPLAN, pages 304–317. ACM, Oct. 1998.

[24] G. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 17(2):29–36, Aug. 1997.

[25] Programming Systems Group, Swedish Institute of Computer Science, Sweden. *SICStus Prolog User's Manual*, 1995.

[26] H. Ritch and H. M. Sneed. Reverse engineering programs via dynamic analysis. In *Proceedings of WCRE '93*, pages 192–201. IEEE, May 1993.

[27] M. Sefika, A. Sane, and R. H. Campbell. Monitoring complicance of a software system with its high-level design models. In *Proceedings ICSE-18*, pages 387–396, Mar. 1996.

[28] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[29] TakeFive Software GmbH. *SNiFF+*, 1996.

[30] S. Tichelaar and S. Demeyer. An exchange model for reengineering tools. In *ECOOP'98 Workshop Reader*, LNCS 1543, pages 82–84, 1998.

[31] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow insensitive c++ pointers and polymorphism analysis and its application to slicing. In *Proceedings ICSE '97*. IEEE, May 1997.

[32] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proc. OOPSLA'98*, pages 271–283, 1998.

[33] N. Wilde and R. Huit. Maintenance support for object-oriented programs. *Transactions on Software Engineering*, 18(12):1038–1044, Dec. 1992.

[34] N. Wilde and M. C. Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):1038–1044, 1992.

[35] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, Jan. 1995.

[36] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS USA '98*. IEEE, Aug. 1998.