Using Run-Time Data for Program Comprehension*

Thomas Gschwind Johann Oberleitner Martin Pinzger
Distributed Systems Group
Technische Universität Wien
Argentinerstraße 8/E1841
A-1040 Wien, Austria, Europe
{tom,joe,pinzger}@infosys.tuwien.ac.at

Abstract

Traditional approaches for program comprehension use static program analysis or dynamic program analysis in the form of execution traces. Our approach, however, makes use of runtime-data such as parameter and object values. Compared to traditional program comprehension techniques, this approach enables fundamentally new ways of program analysis which we have not seen so far. Reflection analysis which allows engineers to understand programs making use of reflective (dynamic) method invocations is one such analysis. Another is object tracing which allows engineers to trace and track the use of a given instance of a class within the program to be understood. In this paper, we present these techniques along with a case study to which we have applied them.

1 Introduction

Program comprehension is necessary to get a deeper understanding of a software application. This is necessary if the software application needs to be changed or extended and its original documentation is missing, incomplete, or inconsistent with the implementation of the software application. Source code analysis as performed by Rigi [15] or Software Bookshelf [3] is one approach for program comprehension. These approaches generate a source model that enables the generation of high level sequence and collaboration diagrams. Since the collaboration between different modules also depends on runtime data, dynamic analysis tools such as Software Reconnaissance [2, 16], BEE++ [1] or Form [12] have been developed. These approaches identify the code that implements a certain feature by generating different execution traces.

Since the above approaches only rely on static data and dynamic data in the form of execution traces they cannot be used to perform certain types of analysis. In this paper, we present a new approach that also takes runtime data such as the parameters passed during method invocations or the state of a given program module into account. Taking this data into account enables new kinds of software analysis.

- Reflective (dynamic) method invocations can be identified and understood. Traditional analysis techniques only identify the existence of such a call but are unable to identify the actual method invocation.

- Execution traces can be reduced to those method invocations that pass a given instance of an object. This allows developers to understand how a certain object is being used.

- It simplifies the understanding of a program's threading behavior and how data is exchanged between different threads.

In this paper, we focus on the first two kinds of analysis, and how we have applied these kinds of analysis to Sun Microsystem's Bean Development Kit [13].

The remainder of this paper is organized as follows. In Section 2, we present ARE, our tool for program comprehension. Section 3 describes how we have implemented the analysis of reflective (dynamic) method calls and Section 4 shows how object instances can be traced and how this data can be used for program comprehension. A case study that demonstrates the benefits of the new analysis techniques is presented in Section 5. Future work is presented in Section 6 and related work in Section 7. Finally, we draw our conclusions in Section 8.

[1]We gratefully acknowledge the financial support provided by IBM Research Division, Zurich Research Laboratories in the form of an IBM University Partnership Award and by the European Union as part of the EASYCOMP project (IST-1999-14191).

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

2 ARE

ARE is A Reverse Engineering tool that provides a flexible and extensible approach to gather runtime data and to analyze the dynamic behavior of existing software systems. Both, flexibility and extensibility of ARE is guaranteed by the use of a layered architecture. Engineers can change the configuration of each layer during run-time, except for the low-level instrumentation layer which, however, can be changed during instrumentation time. For instance, they may choose a different filter or analysis algorithm and in this way control the analysis process.

Our architecture strictly separates the primary concerns of our dynamic analysis tool: extraction of the runtime data, filtering and recording the information of interest, and analysis of the stored information and visualization of the results.

The bottom layer contains the modules for the extraction of run-time information. We use AspectJ [8], an aspect-oriented programming [9] approach, to instrument the program to be analyzed. The instrumentation is implemented by weaving a tracing aspect into the program. This aspect allows us to identify each method and constructor invocation and forwards this data to the recording layer. Although our tracing aspect is kept general enough to fit most purposes, the tracing aspect may be tuned. For instance, the instrumentation can be changed to instrument only a subset of the program's classes if performance is of major concern.

The advantages of using AspectJ are that AspectJ instruments the application automatically, does not turn off Java's just-in-time compiler, and enables access to the parameters of method and constructor calls. The latter capability of AspectJ in particular is mandatory to analyze reflective method calls and the flow of object instances passed between different modules of a software application.

Currently, we are using AspectJ 1.0.6 which requires the availability of the application's source code to weave in our tracing aspects. A beta version of AspectJ 1.1, released in November 2002 [6], is able to weave an aspect into a program's class files without requiring access to the application's source code.

The recorder layer is responsible to filter the trace data obtained from the instrumentation layer and to record it in a trace database. To preserve a maximum degree of flexibility recorders can be changed and customized during run-time. For instance, we provide recorders to monitor the construction of objects, to trace the uses of a given object, or to monitor reflective method calls, constructor calls, or field accesses, and adding more such recorders is a trivial task. This allows engineers to control the analysis process and to focus on the portions of interest.

The top layer of ARE provides data-analysis and visualization tools. Concerning the analysis tools described in this paper our focus is on resolving reflective method calls and on the analysis of the flow of object instances. In the context of program comprehension our analysis tools give detailed insights into complex software systems by showing which object really has been invoked and by showing how objects are passed around between different modules.

3 Reflection Analysis

A reflective system provides structures and mechanisms to represent or modify a given system itself [10]. The Java Reflection API [14] provides a set of built-in classes that support this technique. This API can be used to determine the class of an object, all the interfaces implemented by the object, to generate a method call during run-time, and other such features as shown in Figure 1.

```
1  import java.lang.reflect.*;
2  public class ReflectionExample {
3   public static void main(String[] args)
4   throws Exception {
5    Class c=
6     Class.forName("javax.swing.JButton");
7    Object o=c.newInstance();
8    c=o.getClass();
9    Method m=c.getMethod("getText",null);
10   System.out.println(m.invoke(o,null));
11  }
12 }
```

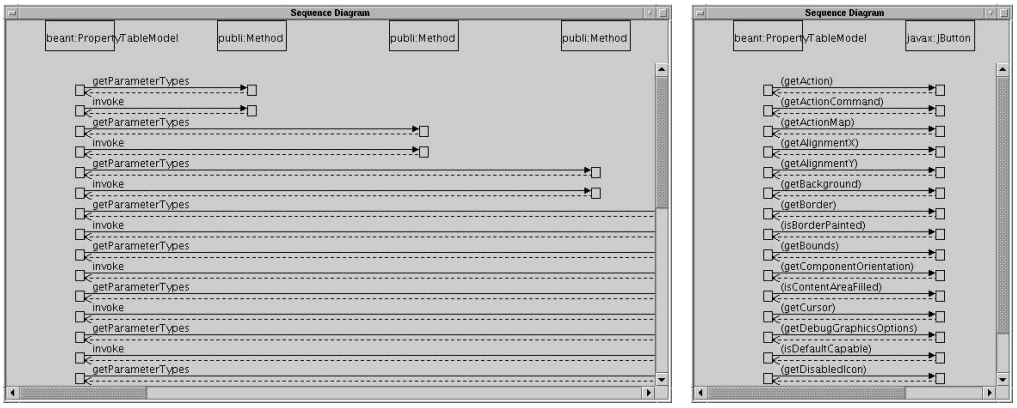
Figure 1. A Reflection Example

The primary class responsible for reflective functionality is the `java.lang.Class` class. Additionally, all Java objects have a predefined `getClass()` method that returns the object's corresponding `Class` object (line 8). The `Class` class allows the programmer to query for and to instantiate classes unknown until run-time (lines 5–7).

Reflection also allows programmers to find out about an object's type superclasses and all implemented interfaces. Another mechanism that the Java reflection API provides is the support for dynamic class member access. This allows programmers to retrieve objects that reify the methods (line 9), constructors, and fields of a class. These objects provide the names, and signatures of the corresponding class members. Unlike pure static representations of class members they support method invocation (line 10), object instantiation, and field access and modification.

This very-late object construction and binding is heavily used within extensible Java applications. In the example above, for instance, the name of the class to instantiate, or the name of the method to invoke could have been supplied by the user of the application. Some programming

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE



(a) Without Reflection Analysis (b) With Reflection Analysis

Figure 2. Sequence Diagram with and without Reflection Analysis

patterns that make use of reflection are common in Java applications. One typical usage scenario is the integration of plugins into an existing application. A third-party plugin implements one or more interfaces predefined by the main application. Since the main application does not know the name of the class implementing the plugin, it is necessary to store the name of the plugin class in a configuration file. Subsequently the plugin can be instantiated with reflection. Other uses of reflection will be shown in Section 5.

Although reflection is indispensable for building dynamic and extensible systems the analysis of reflective systems is difficult. Static analysis does not deliver appropriate results about the classes or methods being used in reflective calls. It is only possible to indicate that the program uses reflection and sometimes to restrict the result space to a set of classes or interfaces. The reason for this is that the name of the method to be invoked is only supplied during run-time and hence, unavailable during static analysis.

For a better understanding of the application, ARE analyzes applications using reflection by combining dynamic analysis with the application's run-time data. During data collection not only the initiating object of a reflective method invocation or object instantiation is recognized but also the called object in case of a reflective method call as well as the instantiated object in case of an object instantiation. ARE replaces the call data of the reflective method invocation with that of the actual call (i.e., the concrete classes, methods or constructors being used during reflective calls), hence providing a more detailed analysis of the program execution.

To show the benefits of this analysis Figure 2(a) shows a sequence diagram generated by using program traces alone with reflection analysis being disabled. Since a `Method` object is created for each method being used by the application's reflective invocations the use dependencies between the calling object, and the methods being called are not directly visible. Instead an object for each method is shown.

Figure 2(b), on the other hand, shows the same sequence diagram but with reflective method calls replaced by the actual methods that have been invoked. The method names are surrounded by parenthesis to indicate that the methods have been invoked using Java's reflection API. This diagram is smaller and easier to understand while showing the same amount of information.

4 Object Tracing and Tracking

The goal of this type of analysis is to monitor how a single instance of a class is being used within the program. This analysis provides a variety of insights about the execution of a given program since it only reports the calls to the object of interest and not just all uses of the corresponding class.

Whenever ARE displays traces of the program's execution, the engineer may select the individual methods, its parameters, caller, or callee shown in the diagram and add them to the list of objects to be monitored. Alternatively, the engineer can inspect these objects by displaying its attributes and can also monitor these attributes.

Object tracing is available in two different forms. In the first form, we simply record each method invocation where the object to be monitored is either the caller or the callee. In the second form, we track the objects to be monitored. Hence, whenever an object of interest is passed as an argument of a method call, this method call is being recorded.

Object tracking poses several challenges. It is not sufficient to simply check each individual parameter of the method invocation, since the object of interest might be encapsulted within one of the method's parameters. For in-

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

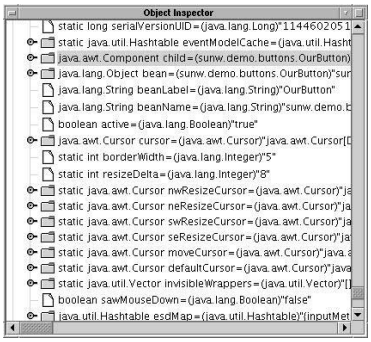


Figure 3. The Bean Development Kit's `Wrapper` Class

stance, the object of interest could be passed via the `child` attribute of the component `Wrapper` class shown in Figure 3.

To identify such calls, we analyze the structure of each object passed during a method invocation. In our initial approach, we performed a complete analysis of each object which noticeably slowed down the execution of the program. To solve this problem, we added an additional parameter indicating how deeply objects have to be analyzed. While experimenting with this parameter, we have found out that not all objects which are passed as part of another object are used by the callee. Hence, these calls should not be recorded. This is not even surprising since the object of interest is buried deeply within the parameter's object structure.

For the application we have reverse engineered, we have found out that a recursion depth of 5 provides an adequate tradeoff. This level allows the program to execute at an responsive level, provides a reasonable tracking accuracy and weeds out most of the unwanted calls.

Besides cutting off after a fixed level of recursion, we are also experimenting with a more advanced tracking algorithm that takes more execution data into account. Instead of inspecting each individual parameter, we monitor how a method uses the parameters that have been passed to it.

5 Case Study

To demonstrate the usability of our approach, we have analyzed Sun Microsystem's Bean Development Kit (BDK) [13], a program that serves as a test environment for JavaBeans components [4]. The BDK allows its users to instantiate JavaBean components and provides a graphical representation of the components' properties. Additionally, the BDK allows its users to bind events fired by one component to a method call of another component. Since the BDK is designed to work together with practically any third party JavaBean component it uses reflection to determine the component's properties.

The first aspect of the application we have considered is the instantiation of a component. To get hold of the implementation of this functionality, we have used ARE's `ConstructorRecorder` which records object instantiations no matter whether they have been instantiated with Java's `new` operator or using reflection. After we have started the BDK, we instantiated an `OurButton` component. The sequence diagram we have obtained (Figure 4(a)) shows that the component was instantiated using reflection by a `JarInfo` object.

Additionally, shortly after instantiating the component, the BDK instantiates a `Wrapper` class. The object inspector (also shown in Figure 4(a)) shows that the `OurButton` component is passed to the `Wrapper` class as a parameter of the object's constructor. After inspecting the `Wrapper` class and the instantiation of another component, it got apparent that the BDK instantiates one such class for each component. As it turned out, this class is used for the management of the individual components (e.g., such as taking care of the components that have been changed).

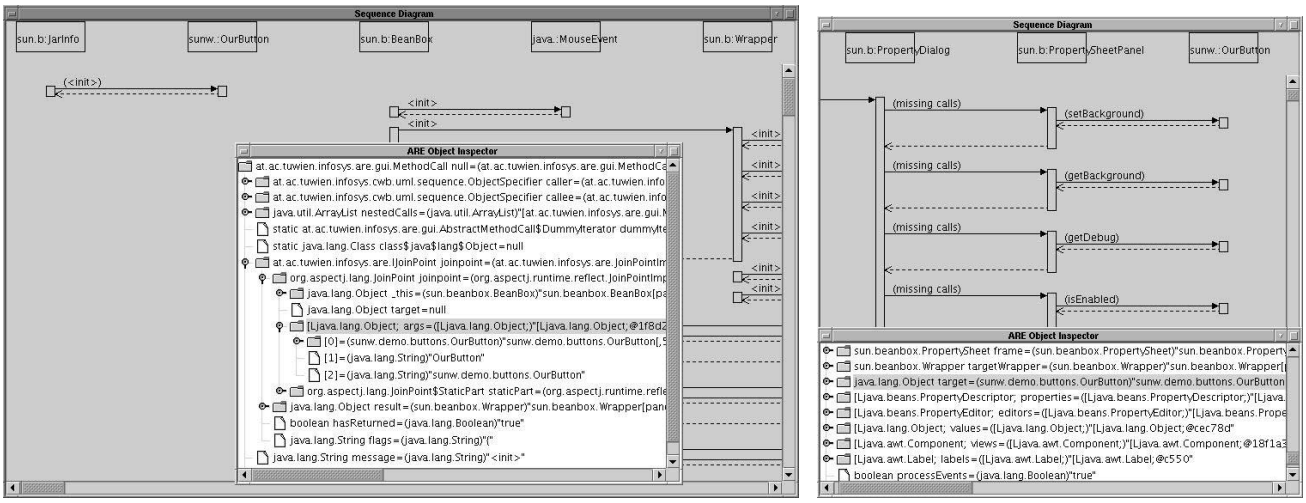
The second aspect we have investigated was changing a component's property, such as the button component's background color. For this test we used ARE's `InstanceTracer` to trace the uses of the button component, we have instantiated previously. The advantage of using this recorder is that the `InstanceTracer` only records activities relating to this instance and not to any other button components used as part of the BDK itself.

The trace obtained with this recorder is shown in Figure 4(b). Interestingly, the second call recorded is already a reflective call to the component's `setBackground` method. The first call was the construction of the `PropertyDialog` object. The calls in between these two calls have not been recorded since they were unrelated to our button component. Hence, these calls are labeled as missing. At first we were unsure whether our recorder missed one of the interesting method calls such as querying the component for its current background color. After a short look at BDK's property window, however, it was obvious why the button had not been queried for its background attribute. The button's background color was already displayed in the BDK's property window and hence was passed to the property editor directly. The subsequent `get...` invocations are simply used to redisplay all the component's properties in the property window.

6 Future Work

The current version of ARE uses a relatively simple algorithm to track objects passed between different modules. As mentioned in Section 4, however, we have already a pro-

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE



(a) Component Instantiation

(b) Property Change

Figure 4. Analysis of the Bean Development Kit

totype implementation for a more advanced object tracking algorithm. One minor drawback of this algorithm is that, unlike our current implementation, it cannot do object tracking across modules whose source code is unavailable. Hence, more work to find a hybrid approach between these two algorithms is necessary.

ARE comes with some support for understanding the threading behavior of a program. Unfortunately, gaining some understanding of a program's threading behavior is only possible through the use of ARE's console window. From an end-user perspective, however, this is tedious. Hence, we plan for a tighter integration of this kind of analysis. Additionally, we currently do not make use of all the data available for a program's thread analysis and hence, further research is necessary to exploit the full potential of this kind of analysis.

Finally, ARE focuses on the use of run-time data for program comprehension. Although using run-time data provides viable data for program understanding, we believe that it can be combined with existing analysis techniques [2, 5, 11] to build a more powerful and more integrated tool for program comprehension.

7 Related Work

Many program comprehension approaches that exist today integrate static and dynamic analysis techniques. Examples of such approaches are Dali [7], Rigi [15], or the Software Bookshelf [3]. Although, the objectives of our approach are related to these other approaches, we address more specific run-time analysis problems that concentrate on resolving reflective method calls and object traces. In the area of dynamic analysis several tools exist that, however, differ in the area of analysis and the type of instrumentation that they use.

Software Reconnaissance [16] uses test cases as probes to locate code for a particular product feature. The program is instrumented similar to instrumentation for test coverage. Afterwards, two different execution runs are started. In the first run a few test cases are applied that exhibit the desired feature. In the second run other test cases are used that do not use this feature. The difference of the trace files generated by the instrumented code shows which code has been executed by a test for a particular feature. Hence, it is possible to build a mapping between features and code. An extension of Reconnaissance that supports concept analysis was presented in [2]. Concept analysis is used to identify the most feature-specific subprograms among all executed subprograms. A static analysis uses this subprogram to identify additional feature-specific subprograms along a dependency graph. While Reconnaissance provides support for well established analysis techniques, it does not allow engineers to gain information about reflective method calls and object traces as provided by ARE.

BEE++ [1] is a C++ based object-oriented framework for the dynamic analysis of distributed systems. BEE++ considers the execution of a distributed system as a stream of events. Hence, BEE++ allows the customization of events and event views. Event and event view customization rely on the inheritance mechanisms of C++. BEE++ has a rich object model that separates event generation and event interpretation. The main drawback of BEE++ is that the instrumentation of the application has to be performed by the

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

programmer manually.

Richner et al. [11] presents an iterative and interactive approach to analyze execution traces of object-oriented software systems with respect to the collaborations between classes. For each method invocation the name of the invoked method, sender and receiver class, as well as their identities are recorded. Through pattern matching they find similar execution traces and represent program behavior in terms of collaboration patterns. By querying and browsing this information engineers can focus on specific aspects of the application and wade through a lot of trace information. Whereas this approach groups and abstracts execution traces we focus on the run-time aspect of the analysis. Hence, it would be interesting to look at a combination of both approaches for future versions of ARE.

8 Conclusions

The contribution of this paper is to use the state of object instances during run-time for program comprehension. This kind of data allows engineers to perform new kinds of program analysis. In this paper, we have presented two analysis techniques that have been based on this approach: reflection analysis and object tracing.

Reflection analysis allows engineers to identify and to resolve reflective or dynamic method calls (i.e., identify the actual method and object that have been called). Such method calls typically used by highly extensible applications cannot be resolved using static program analysis. Object tracing allows engineers to analyze the use of a specific instance of a class within a program. Typical applications for this are to track how an object is passed from one module to another or to eliminate calls to other instances of the same class and hence to reduce the size of the program trace to be understood.

To show the usability of the analysis techniques we have developed, we have reverse engineered a JavaBeans composition environments. As we have shown reflection analysis and object tracing have been used extensively throughout this study. Without these new kinds of analysis we would not have been able to understand and compare these two environments that quickly.

References

[1] Bernd Bruegge, Tim Gottschalk, and Bin Luo. A framework for dynamic program analyzers. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 65–82. ACM Press, 1993.

[2] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the International Conference on Software Maintenance*. IEEE, November 2001.

[3] Patrick J. Finnigan, Richard C. Holt, Ivan Kalas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kenny Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

[4] Graham Hamilton. *JavaBeans*. Sun Microsystems, 1.01 edition, July 1997.

[5] Mark Harman and Sebastian Danicic. Amorphous program slicing. In *Proceedings of the IEEE International Workshop on Program Comprehension (IWPC'97)*, pages 70–79. IEEE, May 1997.

[6] Erik Hilsdale, Jim Hugunin, Wes Isberg, Mik Kersten, and Gregor Kiczales. The AspectJ homepage. http://www.aspectj.org/, December 2002.

[7] Rick Kazman and S. Jeromy Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, 1999.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP'01)*, pages 327–355. Springer-Verlag, 2001.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object Oriented Programming (ECOOP'97)*, pages 220–242. Springer-Verlag, 1997.

[10] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 147–155. ACM Press, December 1987.

[11] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of the International Conference on Software Maintenance*, pages 34–43. IEEE, 2002.

[12] Timothy S. Souder, Spiros Mancoridis, and Maher Salah. Form: A framework for creating views of program executions. In *Proceedings of the International Conference on Software Maintenance*. IEEE, November 2001.

[13] Sun Microsystems. The JavaBeans Development Kit 1.1. http://java.sun.com/products/javabeans/software/-bdk_download.html, April 1999.

[14] Sun Microsystems. Java reflection. http://java.sun.com/-j2se/1.4/docs/guide/reflection/, 2002.

[15] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, December 1994.

[16] Norman Wilde and Christopher Casey. Early field experience with the software reconnaissance technique for program comprehension. In *Proceedings of the International Conference on Software Maintenance*. IEEE, 1996.

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE