A Generic Instrumentation Framework for Collecting Dynamic Information

Anil Chawla and Alessandro Orso
College of Computing
Georgia Institute of Technology
{anil|orso}@cc.gatech.edu

ABSTRACT

Performing empirical research in software testing involves executing a set of subjects against one or more test suites and measuring some characteristics of these executions. Such measures are often collected using ad-hoc instrumentation, by inserting probes in the code that collect and report dynamic information at run-time. Another possible approach is to collect the needed information by leveraging capabilities of the runtime system. Both these approaches usually result in measurement tools that are not flexible and are, thus, hard to reuse and modify. To address this problem, we present a generic framework for collecting information on the runtime behavior of a Java program. The framework allows for easily collecting different kinds of dynamic information for a set of executions of the program, such as coverage and profiling of various code entities and program traces at different levels. The framework also lets users easily define how to process the collected information. In the paper, we also present a case study that we performed to evaluate the framework, and that shows its effectiveness and efficiency.

1. INTRODUCTION

Empirical research in dynamic analysis in general, and testing in particular, requires the gathering of different kinds of information on the runtime behavior of a program when run against a set of inputs (or test cases). Examples of such information are coverage data for various code entities, such as statement and branches, traces at different levels of abstraction, and program profiles.

For Java programs, two common approaches for collecting such information are to add to the code ad-hoc instrumentation using a bytecode rewriting library or to leverage capabilities of the runtime systems (e.g., the Java Virtual Machine profiling or debugging interface [7, 8]). Unfortunately, these approaches are usually expensive and result in experimental infrastructure that is not flexible and, thus, hard to reuse and modify. Another approach for alleviating the problem of collecting runtime information is to use an aspect-oriented language. Although aspect-oriented languages provide a convenient mechanism for inserting probes at specific points in a program, they are often inadequate: first, existing aspect-oriented languages are not able to provide certain kinds of information, such as information at the basic-block level; second, aspects can incur a large time overhead.

To solve these problems, we present an efficient, generic framework for gathering dynamic information for Java programs. The framework lets users collect runtime information with limited effort because it allows for easily specifying (1) which types of entities should be monitored at runtime, (2) in which parts of the code such entities should be monitored, (3) what kind of information should be collected for each entity, and (4) how to process the information collected.

The paper also describes a prototype implementation of the framework and an empirical study that we performed to evaluate our approach. In the study, we use the implementation of our framework to perform method tracing and compare our approach with an approach based on the use of an aspect-oriented language in terms of efficiency.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

2. APPROACH

Our goal is to provide an extensible, configurable, and intuitive framework for gathering information from an executing program. Examples of this type of information include coverage, profiling, and data values from specific points in a program's execution. Furthermore, we would like our framework to provide the information that it gathers in a generic manner. Such a capability lets users easily build tools and experimental infrastructure using the framework.

Our framework has two main characteristics:

- It provides a large **library of probes** to collect different kinds of information for different code entities.
- It lets the user define **instrumentation tasks**, which provide a simple way to instrument different entities in different parts of the code, collect different information from the different entities, and process the information in a customized way.

2.1 Library of probes

Each type of instrumentation corresponds to a specific code construct. The set of code constructs that we instrument includes, among others, method calls, field accesses, acyclic paths, and catch blocks. We refer to the code constructs that we can instrument as *instrumentable entities.*

Instrumentable entity	*Information available*
Method entry	enclosing object argument objects
Method exit	return object or exception object
Before method call	target object parameter objects
After method return	return object or exception object
Field read	field object containing object
Field write	old field object new field object containing object
Start of basic block	*none*
End of basic block	*none*
Before a branch	*none*
After a branch	*none*
Throw	exception object
Catch	exception object
Predicate	evaluated predicate result
Acyclic path	*none*

Table 1: Instrumentable entities and information available for each entity.

For each instrumentable entity, our framework can provide a variety of information associated with that entity. For example, in the case of a method call, we can report the target object and all of the parameters passed to the called method. Table 1 shows a complete list of the instrumentable entities in our framework, along with the information available at each point. (We identified the set of instrumentable entities and the information to collect for each of them based on a survey conducted among researchers in the area of dynamic analysis.)

Our concept of instrumentable entities is similar to the idea of joinpoints in AspectJ [1]. However, our instrumentable entities, unlike AspectJ's joinpoints, are specialized for collecting dynamic information, rather than for extending a program's functionality. In particular, there are no AspectJ joinpoint counterparts for certain instrumentable entities, such as basic blocks, predicates, and acyclic paths. Therefore, collecting information for these entities using AspectJ is complex or not possible at all. Moreover, as we shall show in Section 4, our framework collects dynamic information very efficiently, whereas AspectJ imposes a considerable overhead on the program execution. The reason for these differences is that aspect-oriented languages, such as AspectJ, are designed to add to the code crosscutting concerns, such as error handling, and are optimized for that goal. Conversely, our framework is designed and optimized to collect and report runtime information useful for dynamic analysis. Whereas AspectJ collects a considerable amount of information at joinpoints even if only part of the information is needed, our framework collects only the information specified by the user, so limiting the overhead associated with the collection of the dynamic information.

2.2 Instrumentation Tasks

In general, when collecting dynamic information about a set of executions, we are interested in collecting information for some specific entities in the code (e.g., method calls and paths) and in a subset of the program (e.g., in a specific module or set of modules). Our framework lets the user specify, using instrumentation tasks, (1) the types of entities to instrument, (2) the parts of the code in which those entities must be instrumented, and (3) the kind of information to collect from the different entity types, and (4) how to process the information collected.

An *instrumentation task* for a given system is divided into two main parts. The first part specifies the part of the system being instrumented, in terms of classes or methods. The second part specifies which types of entities must be instrumented, what kind of information should be collected from each entity type, and how such information should be processed. An instrumentation task can also, in turn, consist of a set of instrumentation tasks. Composite instrumentation tasks give users the possibility to collect different kinds of information from different parts of the system (e.g., branch coverage for some modules and exception coverage for other modules). Composite instrumentation tasks also allow for processing differently the information collected for different parts of a system.

Because users can precisely specify the kind of information to collect for a given entity type, our framework can reduce the amount of instrumentation, thus optimizing it and reducing its overhead. The kinds of information available for a given type are listed in the second column of Table 1. As the table shows, different types provide different kinds of information. For a method entry, for example, the framework can collect the list of parameters of the call, whereas for a method exit it can collect the return object.

Users can specify how the collected information is processed by declaring which monitor should process which type of information. Monitors are classes that consume the information reported by the probes inserted in the code. Each monitor class is responsible for collecting information reported from a single type of instrumentable entity. For example, a monitor for method-entries will collect information reported from all entry points of methods, whereas a monitor for method-exits will collect information reported from all exit points of methods. Monitors serve as the foundation for measuring tools built on top of our framework. Users can either use the standard monitors provided with the framework, or create their own specialized monitors, by simply implementing an interface provided by the framework.

3. TOOL

InsECT (Instrumentation, Execution, and Coverage Tool) is our implementation of the dynamic analysis framework described in the previous section. InsECT is written in Java and instruments programs at the byte-code level using the Byte-Code Engineering Library [2]. Instrumentation tasks are specified using XML files. The tool leverages a Java analysis tool developed within our group to perform various data-flow, control-flow, and type-inference analysis. The results of these analysis are used to optimize the instrumentation (e.g, by performing virtual-call resolution and leveraging dominance information).

All components in InsECT relate either to instrumentation, monitoring, or analysis. The instrumentation components are responsible for inserting probes and maintaining a centralized ID system. The monitoring components consist of specific monitors for each type of instrumentable entity, as

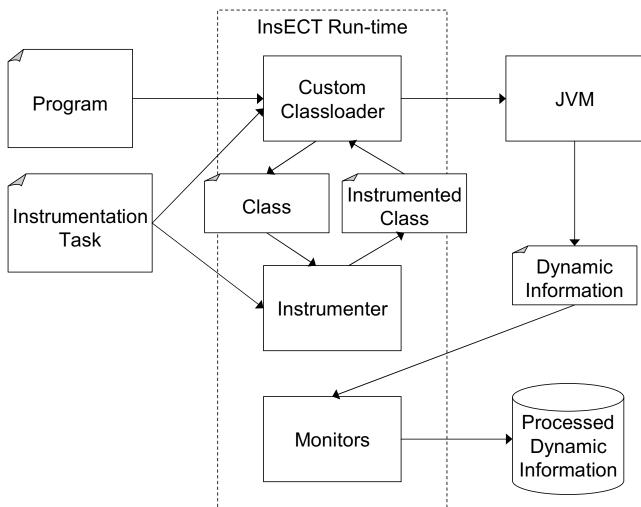

Figure 1: High-level view of InsECT in dynamic mode.

well as a generic monitor capable of collecting and recording the information provided by any probe. Finally, InsECT includes a set of classes that can be used as a foundation for building dynamic analysis and measuring tools. These classes provide methods to retrieve the dynamic information stored by the generic InsECT monitor. These classes also provides methods to access various information about the instrumented program, such as mappings from entities' IDs to the location of the entities in the code.

InsECT can operate in two modes: static and dynamic. In *static mode*, InsECT inputs the program to be instrumented and an instrumentation task, and generates a suitably instrumented program. In dynamic mode, InsECT instruments the code dynamically, when it is loaded at run-time, using a specialized classloader. The dynamic mode allows for more flexibility when the instrumentation must change frequently, possibly dynamically, and users do not want to store a number of different versions of the instrumented program.

Figure 3 shows a high-level view of InsECT when it operates in dynamic mode. As the figure shows, the tool uses a custom class-loader that invokes InsECT's instrumenter on classes that must be instrumented, according to the specified instrumentation task. (Classes that must not be instrumented are loaded normally.) As the JVM executes the instrumented parts of the program, dynamic information is reported to the monitors in the framework or to the custom monitors provided by the user. These monitors process and store the dynamic information.

4. EMPIRICAL EVALUATION

To perform a preliminary validation of our framework, we performed a case study using our prototype implementation. The overall goal of the study is (1) to assess whether InsECT can easily be used for the task of building a measuring tool, and (2) to compare the performance of InsECT to the performance of AspectJ for the same task. More precisely, in the case study, we investigated the following research questions:

1. How difficult is it to build a measurement tool using our framework, compared to building the same tool using AspectJ?
2. How efficient are measurement tools built using our framework, compared to the same tools built using AspectJ?

To address these questions, we used InsECT and AspectJ to build two tools that compute two different kinds of dynamic information:

- *Program traces at the method level:* This tool, that we call $PTrace$, collects traces that consist of one item for each method entry (the method's ID) plus a special symbol for each method exit.
- *Program traces at the method level with parameter information:* This tool, that we call $PTraceParam$, collects the same information as $PTrace$ plus information on the target and parameters of each call.

4.1 Setup

The independent variable in this case study is the approach we used to build tools $PTrace$ and $PTraceParam$: InsECT or AspectJ. Our dependent variables of interest are (1) the time required to build the tools, and (2) the time overhead imposed by the two tools for collecting the dynamic information.

To address the first research question, we selected a student familiar with both InsECT and AspectJ, asked the student to develop the two tools using the two approaches, and measured the time required to build each tool with each approach.

To address the second research question, we used the four tools developed by the student to collect dynamic information for a subject and a set of test cases. As a subject for our study we utilized Jaba (Java Architecture for Bytecode Analysis [3]), a Java analysis tool developed within our research group at Georgia Tech. Jaba consists of about 60KLOC and 530 classes. As test cases, we used the actual Jaba regression test suite, which consists of 707 test cases. We ran the set of test cases on the original Jaba, and on Jaba instrumented by each of the four tools. For each test case, we measured the time to run the test case on the uninstrumented version of the program and on each of the four instrumented programs. Then we used such measures to compute the percentage of overhead for each test case and for each of the four instrumented versions with respect to the uninstrumented version. Finally, we averaged the overhead over all of the test cases, to get a unique value for each of the four tools.

4.2 Results

It took the student about 10 minutes to build both $PTrace$ and $PTraceParam$ using our framework. The time required to build the same tools using AspectJ was about 40 minutes. The case study was performed for two tools and involving only one developer. Therefore, we obviously have no guarantee that these results would generalize to the construction of other tools or to other developers. Nevertheless, the results are encouraging.

The results for the second part of the study also show considerable differences between the two approaches. These results are shown in Table 2. In the table, we refer to the tools built using InsECT as $PTrace_i$ and $PTraceParam_i$, and to

the tools built using AspectJ as $PTrace_a$ and $PTraceParam_a$. Each row in the table corresponds to a tool and reports the time overhead imposed by that tool (i.e., by the instrumentation added by that tool) on the execution of the subject program. The overhead is computed with respect to the execution of the uninstrumented version of the program, as explained in the previous section.

	Time Overhead
$PTrace_i/Uninst$	8%
$PTrace_a/Uninst$	26%
$PTraceParam_i/Uninst$	48%
$PTraceParam_a/Uninst$	86%

Table 2: Time overhead for the tools built using InsECT and AspectJ.

As the table shows, the instrumentation tools built using InsECT impose a considerably lower overhead than the tools built using AspectJ. Also in this case, this is just a case study, and the results may not generalize. However, we have reasons to expect InsECT to be more efficient than AspectJ in general. As explained in Section 2.1, we expect our framework to be more efficient because our approach, unlike AspectJ, optimizes the instrumentation based on the information required by the user and minimizes the instrumentation accordingly.

5. RELATED WORK

AspectJ is an implementation of an aspect-oriented language extension for Java [1]. Central to AspectJ are the concepts of joinpoint, pointcut, and advice. A joinpoint refers to a specific point in the code at which a user can implement functionality. A user of AspectJ can create a pointcut by selecting a set of joinpoints and implementing a piece of code, called advice, to be run at each of those joinpoints. Hence, given the pointcut (set of joinpoints) referring to all method calls, the user can implement an advice that reports all of the information about that method call. AspectJ, although powerful for adding crosscutting functionality to the code, is limited in its library of joinpoints and is incapable of gathering certain types of information needed by some dynamic analyses.

The *Java Instrumentation Engine* (JIE) is a generic system for source-code instrumentation [5]. The JIE operates based on an instrumentation configuration. This configuration describes points in the source-code to instrument, and the action to perform at each of these points. Although the tool is fairly generic, operating at the source level makes instrumentation of certain code constructs difficult. Most notably, the JIE is incapable of basic block or method invocation instrumentation. The limitations stem from the lack of global type information.

The *Java Runtime Anaysis Toolkit* (JRAT) is a static bytecode instrumenter intended for gathering runtime data and metrics for a program [6]. JRAT instrumentation consists of a wrapper method for each method in the program. Each wrapper method gathers information regarding the invocation of its encapsulated method and fires events. JRAT provides a Service Provider Interface for event-handling. The main limitation of JRAT is that it can only collect very limited dynamic information.

The *Java Instrumentation API* (JIAPI) is a framework for bytecode instrumentation [4]. JIAPI is capable of both static and dynamic instrumentation, and provides abstractions for bytecode manipulation. Instrumentation with JIAPI consists of chaining several instrumenters together. Each instrumenter manipulates the instruction list of a method and then forwards the instruction list to the next instrumenter in the chain. In this manner, JIAPI enables arbritary instrumentation of the bytecode. Data is collected from the instrumentation through the use of events. Currently, the event handling interfaces are limited to methods, fields, and exception events.

6. CONCLUSION

In this paper, we presented our generic framework for collecting dynamic information for Java programs. The framework can be used to build infrastructure to support experiments for various dynamic analyses, including testing. We also presented some preliminary results that show that using our framework (1) can be simpler than using alternative approaches, and (2) can produce measurement tools that are more efficient than tools built using alternative approaches.

Acknowledgments

This work was supported in part by NSF awards CCR-0096321, CCR-0205422, CCR-0306372, and SBE-0123532.

7. REFERENCES

[1] Aspectj project. `http://eclipse.org/aspectj/`.

[2] Byte-Code Engineering Library (BCEL). `http://jakarta.apache.org/bcel/`.

[3] Java Architecture for Bytecode Analysis (JABA). `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`.

[4] Java Instrumentation API (JIAPI). `http://jiapi.sourceforge.net/`.

[5] Java Instrumentation Engine (JIE). `http://www.forum2.org/eran/jie/`.

[6] Java Runtime Analysis Toolkit (JRAT). `http://jrat.sourceforge.net/index.html`.

[7] Java Virtual Machine Debugging Interface (JVMDI). `http://java.sun.com/j2se/1.3/docs/guide/jpda/jvmdi-spec.html`.

[8] Java Virtual Machine Profiler Interface (JVMPI). `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html`.