InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse

Arjan Seesing and Alessandro Orso
College of Computing
Georgia Institute of Technology
a.c.seesing@ewi.tudelft.nl, orso@cc.gatech.edu

ABSTRACT

The heterogeneity and dynamism of today's software systems make it difficult to assess the performance, correctness, or security of a system outside the actual time and context in which it executes. As a result, there is an increasing interest in techniques for monitoring and dynamically analyzing the runtime behavior of an application. These techniques are usually implemented using ad-hoc solutions that result in tools that are hard to develop, maintain, and reuse. To address this problem, we present a generic framework that enables the collection of different kinds of runtime information, such as data about the execution of various code entities and constructs. The framework lets users easily define how to process the collected information and is extensible to collect new types of information. We implemented the framework for the Java language as a set of Eclipse plug-ins. The plug-ins provide an intuitive GUI through which the functionality of the framework can be exploited. We also present an example of use of our framework and plug-ins that helps illustrate their functionality and shows their ease of use.

1. INTRODUCTION

Today's software systems run in heterogeneous and dynamic environments, and are themselves increasingly dynamic in nature. Heterogeneity of runtime environments can cause different software behavior in different contexts, and the behavior of different instances of the software may also vary because of software's dynamic nature. This heterogeneity and dynamism make it difficult to assess the performance, correctness, or security of a software system outside the actual time and context in which it executes. As a result, there is an increasing interest in techniques that allows for monitoring and dynamically analyzing information about the runtime behavior of an application.

For Java programs, two common approaches for collecting dynamic information are to add ad-hoc instrumentation to the code using a bytecode rewriting library or to leverage capabilities of the runtime system (e.g., the Java Virtual Machine Profiling or Debugging Interface [8, 10]). Unfortunately, these approaches are usually expensive, and the resulting tools are hard to develop, maintain, and reuse in different contexts. Another approach that has been used to address the problem of collecting runtime information is to leverage an aspect-oriented language, such as AspectJ [1]. Although aspect-oriented languages provide a convenient mechanism for inserting probes at specific points in a program, they are often inadequate when used for monitoring and dynamic analysis: first, existing aspect-oriented languages are not able to provide certain kinds of information, such as information at the basic-block level; second, aspect-oriented languages are not easily extensible; third, many implementations of aspect-oriented languages tend to be inefficient.

To address these limitations of existing approaches, we defined a generic framework that enables the collection of various kinds of runtime information, such as data on the execution of various code entities (e.g., branches and paths) and constructs (e.g., assignments). Users can collect such information with limited effort because the framework lets them easily specify (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. Moreover, our framework lets users define how to process the collected information and can be extended to collect new types of information.

We implemented the framework for the Java language as a set of Eclipse plug-ins that are freely available (see Section 6). The plug-ins provide an intuitive GUI and wizards through which the functionality of the framework can be exploited. In the rest of the paper, we describe the framework and the plug-ins in detail. We also present an example of use of our framework and plug-ins that helps illustrate their functionality and shows their ease of use.

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. Further, 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: (1) it provides a large **library of probes** for collecting different kinds of information for different code entities; and (2) it lets users define **instrumentation tasks**, which allow for instrumenting 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

Our framework includes a predefined set of probes. A probe is typically associated with a program construct (e.g.,

Instrumentable entity	*Information available*
Method entry	enclosing object argument objects
Method exit	return or exception object
Before method call	target object parameter objects
After method return	return or exception object
Field read	field object containing object
Field write	old field object new field object containing object
Basic block	*none*
Before or after a branch	*none*
Throw or catch	exception object
Acyclic path	*none*
Assignment	value being assigned

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

a method call or the catching of an exception). We refer to the code constructs that we can instrument with probes as *instrumentable entities*. For each instrumentable entity, our framework can provide various 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 partial list of instrumentable entities, along with the information available for each entity. To identify the set of instrumentable entities and the information to collect for each of them, we conducted an informal survey among colleagues working in the area of dynamic analysis.

Instrumentable entities are similar in spirit to joinpoints in AspectJ [1], but are different in four respects. First, instrumentable entities let users fine tune the kind of information collected for each entity, so improving the efficiency of the framework. (In aspect-oriented programs, a considerable amount of information is always collected at joinpoints, even if not used.) Second, our set of instrumentable entities is more extensive than joinpoints. For example, there are no joinpoint counterparts for entities such as basic blocks, predicates, and acyclic paths. Collecting information for these entities using AspectJ is either complex or not possible at all. Third, our set of instrumentable entities is extensible. Fourth, and on a more general note, instrumentable entities, unlike AspectJ's joinpoints, are specialized for collecting dynamic information, rather than for extending a program's functionality.

2.2 Instrumentation Tasks

In general, when collecting dynamic information, we are interested in collecting such 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 this information in the form of an instrumentation task. More precisely, an *instrumentation task* specifies (1) which instrumentable entities to instrument, (2) the parts of the code in which those entities must be instrumented, (3) the kind of information to collect from the different entity types, and (4) how to process the information collected.

Because users can precisely specify the kind of information they want to collect for a given entity type, our framework can reduce the amount of instrumentation needed, thus reducing its overhead. As Table 1 shows, different instrumentable entities 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 or the propagating exception (if any).

Users can specify how the collected information is processed by defining a monitor (or by using an existing monitor) that will receive the information and suitably handle it. Each monitor is a concrete class that consumes information reported by one or more probes inserted in the code (i.e., for one or more instrumentable entities). For example, a monitor for def-use analysis could collect information reported for both field writes and reads. For the users' convenience, our framework includes an initial set of monitors for some common analysis tasks (e.g., to collect various kind of coverage or profiling information). The user can easily define additional monitors for other tasks.

3. IMPLEMENTATION

Our framework is implemented for the Java language as a set of three plug-ins for the Eclipse platform. We call the implemented version of our framework InsECTJ (Instrumentation, Execution, and Collection Tool for Java). A first plug-in in InsECTJ, the *core plug-in*, implements the general framework and uses an extension point to expose its functionality to specific probe inserters. A *probe inserter* is an instrumentation module that extends the extension point in the core plug-in and is capable of instrumenting a specific instrumentable entity. Probe inserters are bundled in a second plug-in, called *probe inserter plug-in*. A third plug-in, the *GUI plug-in*, provides the users with a GUI that makes it easy to use the framework. The main part of the GUI are two wizards. The first wizard lets users easily define monitoring tasks by selecting the type of entities to instrument, the parts of the code in which those entities must be instrumented (at the method granularity level or higher for now), and what information to collect for the various entities. The second wizard supports users in creating new monitors, by generating skeleton code that implements the interface for the appropriate probe inserter(s) and that the user can then complete. Each monitor is a class that gets instantiated during the execution of the program.

We implemented the probe inserters using the Java Byte-Code Engineering Library [2], which is already included in Eclipse. To instrument the code, probe inserters leverage the new instrumentation package available in Java 5.0 [5]. It is worth noting that this part of the implementation can also be used as a stand-alone tool that is configured using an XML configuration file. The configuration file provides the information that would otherwise be provided by Eclipse—it specifies which probe inserters are to be used with which monitors and which classes or methods should be instrumented.

By leveraging Java's instrumentation capabilities, our framework instruments programs at load time. Right before a class is loaded, InsECTJ checks whether the class must be instrumented and, if so, invokes the appropriate probe inserters. Each probe inserter instruments some or all of the class's methods by inserting calls to the appropriate monitor

in correspondence of the instrumentable entity for which the probe inserter is defined. For example, the probe inserter for method entries adds instrumentation before the first statement in each instrumented method. The kind of instrumentation added depends on the entity being instrumented and on the information that the user needs to collect. For example, the instrumentation inserted at method entries differs depending on whether the user is interested in collecting information on the parameters passed to the method or not.

On-line, or dynamic, instrumentation has two main advantages. First, it is totally dynamic and, thus, more flexible. Second, it does not require to keep several copies of the program. One problem with on-line instrumentation, though, is that users must pay the cost of instrumenting the code for each execution. To eliminate this problem, InsECTJ also provides off-line, or static, instrumentation. Users can specify that they want to save the instrumented classes and, thus, eliminate the runtime overhead due to the dynamic instrumentation.

Figure 1 illustrates, using a class diagram, a partial design of our infrastructure and how probe inserters are defined. This design allows for extensibility that goes beyond the definition of new monitors. In fact, in case users need to add an instrumentable entity that is not yet included in our set, they can extend the extension point defined by InsECTJ, as shown in the figure. More precisely, new probe inserters must extend the *AbstractProbeInserter* class, which provides methods to simplify the actual instrumentation. The new probe inserter must also define a new monitor interface, which must in turn extend *MonitorObject*. After the probe inserter has been defined, it behaves like any predefined probe inserter, in that its users will have to create, supported by InsECTJ, actual monitors that implement the probe inserter's interface. The next section provides more details on how to extend InsECTJ.

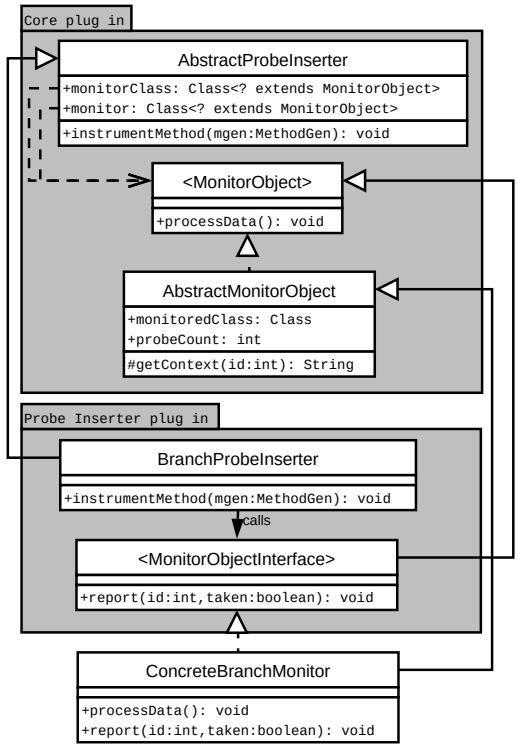

Figure 1: Class diagram that shows how a specific probe inserter (for branches, in this case) extends the core plug-in.

4. EXAMPLE OF USE

Instead of providing complete details on the implementation and functionality of InsECTJ, which would require more than the available space, we illustrate the main characteristics of our framework and plug-ins through an example. In the example, we provide an overview of how a user would use InsECTJ to (1) define a monitor for an existing probe inserter and (2) create a new probe inserter. Section 4.1 describes how to create a monitor and configure the instrumentation and execution of a project using InsECTJ's wizards. Section 4.2 illustrates how the extension mechanism in our core plug-in provides some guidance with the creation of a new probe inserters (in addition, pre-defined probe inserters can be used as examples).

4.1 Defining a New Monitor

This section discusses how to collect information from one or more probe inserters. To this end, we show how to create a new monitor for field reads and writes. Such a monitor could be used, for example, to compute data-flow coverage at the field (or class) level. Users who want to create a monitor would start by creating a module skeleton using the *Monitor Wizard* (see Figure 2). This wizard is implemented as a customized version of the pre-defined *New Java Class Wizard*. We modified this wizard to that `AbstractMonitorObject` is always selected as the superclass for the new monitor, some fields are already filled with appropriate values, and the interface list is specialized to show only relevant interfaces (i.e., probe inserters' interfaces). For our example, the user would select `DefMonitorInterface` and `UseMonitorInterface` as the interfaces of interest, as shown in Figure 2, and the wizard would generate the skeleton code shown in Figure 3.

As the figure shows, the skeleton code contains three method stubs and one constructor. Methods `reportDef` and `reportUse` are the methods defined in the interfaces for the field write and field read probe inserters, respectively. Method `reportDef` (resp., `reportUse`) is called every time a field is defined (resp., used). These methods should be implemented so that they suitably use and/or save the information passed to them about definitions and uses of fields. (The specific use of this information depend on the dynamic-analysis or monitoring task at hand.) Although parameters of the reporting methods are typically specific to a probe inserter, `probeId` is a general parameter that corresponds to an index for accessing various metadata associated with the instrumentable entities (e.g., current-method name and signature and extra context information which can be encoded as a string during instrumentation). Class `AbstractMonitorObject` provides methods to retrieve such metadata (see Section 4.2). This data is inserted into the byte code of the instrumented program, which eliminates the need for external files to store this information. In this way, the instrumented program can run in isolation from the instrumentation framework (with the exception of the monitor classes).

Method `processData` is a method, defined in interface `MonitorObject`, that is automatically called at the end of the execution of the instrumented program by a `ShutdownHook`.

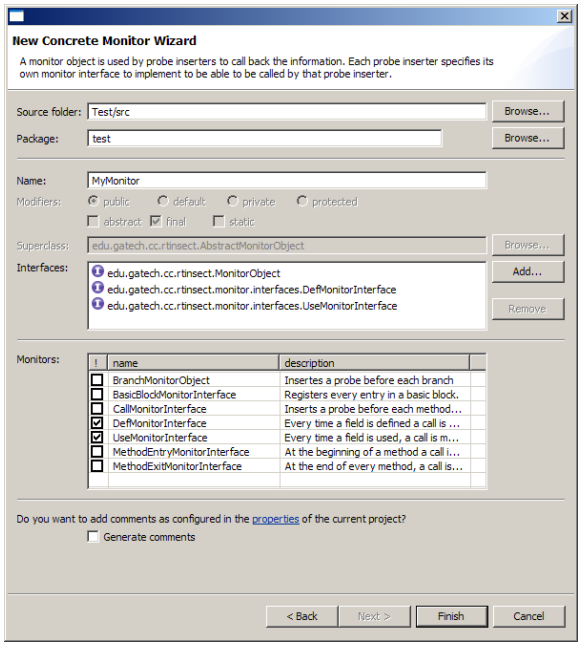

Figure 2: Wizard for creating a new monitor.

```
public final class DefUseMonitor extends AbstractMonitorObject
    implements DefMonitorInterface, UseMonitorInterface {
  public DefUseMonitor(Class parent, int probes, Class monitor) {
    super(parent, probes, monitor);
  }
  public void processData() {}
  public void reportDef(int probeId, Object obj, Object old,
      Object new) {}
  public void reportUse(int probeId, Object obj, Object value) {}
}
```

Figure 3: Skeleton code generated by the monitor wizard for a "def-use" monitor.

The implementation of this method should take care of saving the data collected during execution (e.g., by printing it to the screen, writing it to a file, or saving it into a database). In the future, we plan to add to the framework a call-back mechanism that allows monitors to report information directly to Eclipse. This information could then be shown in views or as context information in the Java editor.

After a monitor has been defined, instrumenting and running a project is a fairly straightforward step that involves the use of our *Launch Wizard*. Figure 4 shows the instrumentation pane in the launch wizard. In the upper part of the wizard's window, the user selects the classes and/or methods to instrument. In the lower part of the window, the user can select which entities should be instrumented in the selected code and which monitors should be associated with the corresponding probe inserters. In future work, we plan to extend this interface to provide more options and give a finer-grained control over the instrumentation.

4.2 Defining a New Probe Inserter

There may be cases in which a user needs to monitor entities that are not in our predefined set of instrumentable entities. In these cases, users can extend InsECTJ by creating a new probe inserter. In this section, we provide a high-level view of the steps that are involved in creating a new probe inserter. For the example, we use branches as

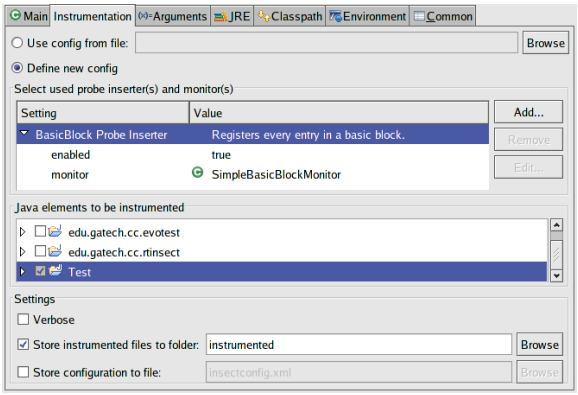

Figure 4: Launching a project with instrumentation using the launch wizard.

```
public interface BranchMonitorObject extends MonitorObject{
  public final static String BRANCH = "branch";
  @InsectInvoke(BRANCH)
  public void reportBranch(int id, int label);
}
```

Figure 5: The monitor interface for a new branch probe inserter.

the instrumentable entity. (Branches are points in the code in which a decision is made about the control flow.)[1] Creating a new probe inserter requires the creation of a new extension of our probe inserter extension point. This extension must specify the following information: (1) the class that implements the probe inserter (this class needs to extend `AbstractProbeInserter`), (2) a unique identifier for the probe inserter, (3) the interface that the probe inserter will call during execution (and that a monitor for the probe inserter must implement), and (4) a short description of the probe inserter as documentation for the probe-inserter users.

For this example, we call our monitor interface `BranchMonitorInterface`. The interface is shown in Figure 5. In method `reportBranch`, parameter `id` provides a handle to retrieve entity metadata, and parameter `label` reports which branch was executed. (This value would be 0 or 1 for `if` or `while` statements, whereas it could assume several different values for a `switch` statement.) The metadata would let users map this information back to the code.

The probe inserter class must implement a method that performs the actual code instrumentation. The framework invokes this method automatically on each method of each class that must be instrumented. In this way, users are relieved from having to deal with a good deal of book keeping and initialization. Moreover, class `AbstractProbeInserter` provides some utility methods that can further simplify the implementation of the probe inserter. To illustrate some of these methods, in Figure 6, we show a simplified implementation of the probe inserter. For example, method `getMonitorObjectInstance` returns an instance of the appropriate monitor for use in the probe. For another example, method `getInvokeInstruction` retrieves the method to be called in the monitor interface using a previously assigned unique name; this name is associated with the class by means of a Java annotation. `AbstractProbeInserter`

[1]Our framework already provides branch coverage, and we selected these entities just for the sake of the example.

```
public void instrumentMethod(MethodGen mgen) {
  InstructionList list = mgen.getInstructionList();
  for (InstructionHandle current = list.getStart();
       current != null; current = current.getNext()) {
    Instruction instr = current.getInstruction();
    if (instr instanceof BranchInstruction){
      InstructionList probe = new InstructionList();
      // push monitor on the stack
      probe.append(getMonitorObjectInstance());
      // push the probe id on the stack
      probe.append(new PUSH(cpgen, getProbeId()));
      // normally we would have to calculate this value
      probe.append(new PUSH(cpgen, 0));
      // execute our monitor method
      probe.append(getInvokeInstruction(BranchMonitorObject.BRANCH));
      ...
      // insert probe
      Util.insertProbe(probe, list, current);
} } }
```

Figure 6: Code for a simplified branch probe inserter.

also provides methods to store extra context information for a probe in the form of a string. Such information can then be retrieved, inside the monitor, using the metadata-access method `getContext`, provided by `AbstractMonitorObject`.

5. RELATED WORK

The *Java Instrumentation Package* [5] is a new package in the Java 5 specification that simplifies the dynamic instrumentation of Java programs. Classes can be completely transformed before being loaded, and smaller changes can be performed even for already loaded classes. Our framework uses this mechanism to instrument classes before they are loaded without the use of a class loader.

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 new joinpoints are hard to add to the language. AspectJ is also intergrated in Eclipse using a set of plug-ins.

The *Java Instrumentation Engine* (JIE) is a generic system for source-code instrumentation [7]. 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, JIE is incapable of basic block or method invocation instrumentation.

The *Java Runtime Anaysis Toolkit* (JRAT) is a static bytecode instrumenter intended for gathering runtime data and metrics for a program [9]. JRAT's 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 limitations of JRAT is that it can only collect very limited dynamic information and is not easy extensible.

The *Java Instrumentation API* (JIAPI) is a framework for bytecode instrumentation [6]. 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, while our framework can collect a wider amount of information.

The *Eclipse Instrumentation Framework* [4] allows developers to collect information about how people are using their Eclipse-based applications by sending back periodic reports to a central server. This is not a generic instrumentation framework, in that it uses instrumentation to record user preferences and statistics, rather than low-level application data.

The *Eclipse Monitor Plug-in* [3] instruments the Eclipse Framework itself to enable a developer to view the interaction between plug-ins and monitor plugins' execution. It also contains an extensible visualization framework.

6. CONCLUSION

In this paper, we presented a generic framework that allows for collecting different kinds of runtime information for a program, such as data about the execution of various code entities and constructs. The framework is implemented in a tool called InsECTJ. InsECTJ consists of as a set of Eclipse plug-ins that make it easy to use the framework for collecting dynamic information for Java programs. The framework is designed to be flexible and easily extensible, so as to be able to accommodate different types of dynamic analysis and monitoring needs. We showed, through an example, how the framework and the plug-ins can be used to track certain events while running a Java program. We also discussed how the framework can be extended to collect information for additional code entities. InsECTJ is available for download at `http://www.cc.gatech.edu/~orso/software/insectj.html`.

7. REFERENCES

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

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

[3] Eclipse Monitor. `http://www.eclipsefaq.org/chris/monitor/`.

[4] Eclipse Instrumentation Framework. `http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-ui-home/instrumentation/index.html`.

[5] Instrumentation API in Java 5. `http://java.sun.com/j2se/1.5.0/docs/`.

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

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

[8] Java Platform Debugger Architecture (JPDA). `http://java.sun.com/products/jpda/index.jsp`.

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

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