Probing for a Continual Validation Prototype

by

Peter W. Gill

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

____

May 2001

APPROVED:

____

Professor George T. Heineman, Major Thesis Advisor

____

Professor Elke Rundensteiner, Department Reader

____

Professor Micha Hofri, Head of Department

Abstract

Continual Validation of distributed software systems can facilitate their development and evolution and engender user trust. We present a monitoring architecture that is being developed collaboratively under DARPA's Dynamic Assembly for System Adaptability, Dependability, and Assurance program. The monitoring system includes a probing infrastructure that is injected into or wrapped around a target software system. Probes deliver events of interest to be processed by a monitoring infrastructure that consists of gauges for delivering information to system administrators. This thesis presents a classification of existing probing technologies and contains a full implementation of a probing infrastructure in Java.

Acknowledgments

I would like to thank Dr. George Heineman, my advisor, for the opportunity to work on the DASADA project and his encouragement. I would also like to thank Dr. Elke Rundensteiner for being the department reader of this thesis. Finally, I would like to thank Andreas Koeller for providing LaTeX assistance and Sara and Cooper Pratt for their patience while I was working on this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems are becoming more complex in a number of ways. First, many systems are now dynamically reconfigurable and can incorporate components that were unknown and possibly unavailable at design time. Second, code can be generated directly from specifications, making understanding the system implementation more challenging. Third, the scale of large component-based software systems makes them prohibitive to formally analyze. Finally, a system that has been fully analyzed prior to initial deployment will almost certainly change after being deployed. Re-analysis of the entire system would generally not be feasible.

Continual validation (run-time monitoring) provides a solution to help developers understand complex systems. It can augment traditional static analyses, testing and simulation and can detect errors that traditional methods can not. A continual validation system collects, aggregates, and disseminates information about a running system to decision agents (human or otherwise). Many high performance and high assurance systems are built with their own proprietary diagnostic and monitoring facilities. However, the high cost associated with building custom monitoring systems has prevented their widespread use. As part of DARPA's DASADA program [11] researchers from a number of institutions are collaborating to develop generic monitoring technologies. The goal is to provide continual validation to a broad range of distributed component-based systems assembled at least in part from a mix of commercial off-the-shelf (COTS) and open source components.

We have collaborated with researchers at Columbia University to develop a prototype Monitoring Architecture called Kinesthetics eXtreme (KX). The objective of KX is to monitor how well a running system meets its functional and extra-functional goals and constraints. A proof-of-concept implementation of KX was demonstrated at the DARPA DASADA Demo Days in Baltimore Maryland in June 2001.

KX was designed to monitor distributed systems and is itself a distributed system. KX is broken down into 3 primary areas: probe infrastructure, event infrastructure, and gauge infrastructure (see Figure 1.1).

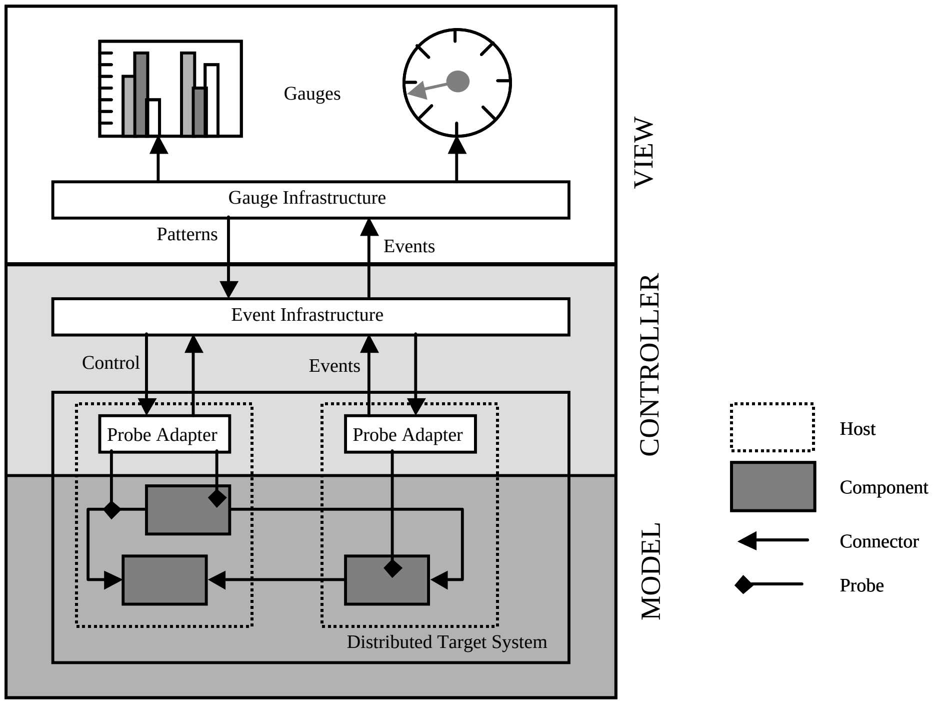


Figure 1.1: A high-level view of the KX monitoring system.

The probe infrastructure includes the various types of probes, the mechanisms for inserting probes into a target system, and policies for how the probes can be controlled by the monitoring system. Probes are used to collect information about running software systems. They detect and reify events in the target system and deliver them to the monitoring system. The goal of probing is to efficiently collect information about a system at run-time.

The event infrastructure is composed of an event standard, an Internet scale event service (SIENA) [5], and persistent storage. An event in the target system can be an actual event passed in an asynchronous messaging architecture, or it can be a method call, a file read or write, or a variable access. Such events offer an appropriate model for understanding system behavior because they capture the interactions between the components of the system.

The gauge infrastructure provides a framework for the creation and execution of gauges. Gauges are event consumers. Gauges subscribe to events from the event infrastructure. They process individual events and search for patterns of events. The primary goal of gauges is to provide information to decision agents (human or automated). In future versions of KX gaugents (gauge + agent) may take an active role in dynamically reconfiguring the target system in response to changing conditions.

KX and the target system can be viewed as an instance of the *model/view/controller* design pattern (see Figure 1.1) [15]. The model is information generated by the execution of the target application. The gauges represent a number of possible view(s) that can be derived from the model's data. The probe infrastructure and event infrastructure make up the controller. The controller serves to decouple the model and the view by delivering information using a publish/subscribe protocol.

1.1 Goals and Evaluation

This thesis has three main contributions.

The first contribution is related to probes. We examine the ways that probes can collect information from running systems and describe a number of existing technologies that can be used as probes. We explicitly describe a number of probes that can be implemented using Active Interfaces [18].

The second contribution is the development of a Probe Run-Time Infrastructure that can be used to deploy and manage probes at run-time. We describe a high-level design that would allow for implementation of the run time infrastructure for the various probe mechanisms. We provide an implementation and demonstration of the design for Active Interfaces.

The third contribution of this thesis is participation in the development of a demonstration of the full monitoring architecture. This demonstration was shown at the DASADA Demo Days conference in June 2001. The target system for the demonstration is GeoWorlds a large component-based system that can script together information sources and processing components dynamically at run-time [9]. The core of the GeoWorlds system is written in Java. We utilized the Active Interfaces Probe Run-Time Infrastructure to support the development of probes to collect information from GeoWorlds at run-time. Additionally we assisted in the development of gauges and meaning full scenarios which highlight the ability of the monitoring system to detect problems in the system.

1.2 Outline

This thesis is divided into six chapters. In Chapter 2 we describe the role of probes in a continual validation system. We discuss mechanisms that can be used as probes and compare the characteristics of the different probe mechanisms. In Chapter 3 we motivate the need for and describe a high level design of a system to deploy and control probes at run-time. Chapter 4 describe an implementation of the high level design for Active Interface probes and present a small proof of concept demonstration of it. In Chapter 5 we describe KX, the entire continual validation system that is being developed collaboratively with researchers at Columbia University. We describe how the Active Interface Probe Run-Time Infrastructure fits in to the overall system and discuss gauges that we constructed for use in the demonstration of the full system. In Chapter 6 we summarize the main points and contributions of this thesis. Additionally we outline future work.

Chapter 2

Probes

2.1 Introduction

KX, the monitoring system described in the introduction is composed of three parts. The probe infrastructure, the event infrastructure and the gauge infrastructure. The probe infrastructure can be further divided into probes and probe control policies. In this chapter we focus on probes. Probes are software that collect information that can be used to develop an understanding of how well a software system is meeting its goals.

In this chapter we begin by describing models of target software systems. We describe the role of probes as they relate to these models.

While it is important to understand the environment in which software systems execute the primary focus of this chapter is on probes that are used to detect events in the target system software. We describe a number of mechanisms that can be used as probes and describe how they function and what benefits and shortcomings result from their use. We then compare the mechanisms with each other discussing the attributes that distinguish them from each other. Finally we develop a taxonomy based on the characteristics of the various types.

2.2 Models of Target Systems

Software architecture provides a high level view of the structure of software systems. Software architecture identifies global control structure, and inter-component communication protocols [17, 27]. Components in this sense can be explicitly developed stand-alone components or cohesive software modules in a legacy system. Components encapsulate data and behavior behind a public interface. Connectors define the interactions between components. Connectors encapsulate communication mechanisms and protocols. They may correspond to actual compilable units (sometimes called connecting-components) or may be a representation for sequences of procedure calls, shared data, pipes, event broadcast or complex protocols [2, 29]. This simplified description of software architecture serves as the basis for a model of target systems.

As described above a target system can be viewed as computational units that communicate to achieve the desired result. To monitor this system, it is necessary to monitor the components, the connectors, and the environment in which the software is running. Probes are software entities that interact with the parts of a system to collect the necessary information.

We use the software architectural view of software to broadly define probes as being component boundary intrusive, connector intrusive or architecturally intrusive (see Figure 2.1). Intrusive implies the location of the probe relative to the structures of the architecture. Component boundary intrusive probes collect information from inside a component boundary. Any modification of the target system to accommodate a component intrusive probe is contained entirely within the component being monitored. Connector intrusive indicates that the probes are localized inside connectors. Connectors can be simple bindings between the methods provided by component or they can be first class objects themselves [29, 2]. In figure 2.1 the block arrows represent actualized connectors. If the connector is a binding, it contains no code and cannot be directly probed. Line arrows represent bindings in the figure. Monitoring must occur inside the components the connector connects or it must rely on indirection (wrapping or otherwise) to intercept control. On the other hand, actualized connectors can be probed directly. Architecturally intrusive probes are visible at the architectural level. This can mean that an entirely new monitoring component is added to the system or that the probed component is seen differently by the other components in the system than was the original component.

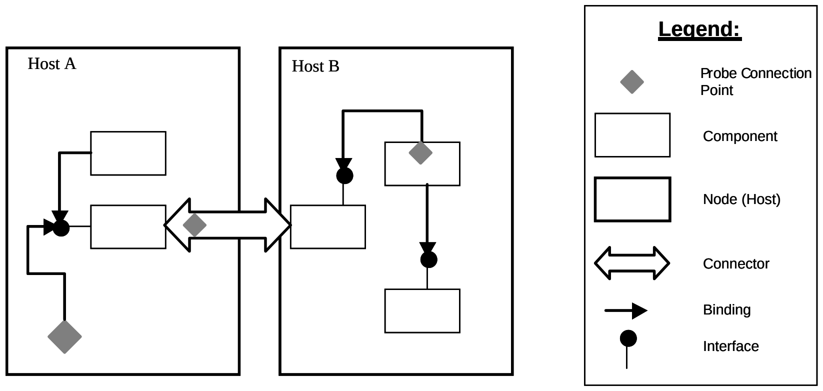


Figure 2.1: Model of Target System.

Probes detect events. Events capture and/or reify communication in the target system. In an asynchronous message passing architecture such as C2 [35] this entails capturing messages in the target system, translating them into the monitoring system's event standard, appending appropriate meta-data and injecting them into the monitoring system. In more traditional software systems, inter-component communication is commonly achieved by method calls or the sharing of data. In these systems events are not first-class objects in the target system. Probes must capture the activity being observed in an event and deliver it to the monitoring infrastructure. Intercepting control before and after method calls or data accesses allows the communication to be detected and described. Events that encapsulate this information can be created and delivered to the monitoring system. Many of the probe technologies presented in this chapter are mechanisms that provide this ability.

All software systems are impacted by the hardware on which they run. When hardware resources are scarce or overloaded, system performance will suffer. This problem is exacerbated with distributed systems due to the greater number of hardware and network resources on which the software system relies. Therefore in addition to monitoring the components and connectors of the software architecture it is important to monitor the non-software resources that are utilized by the software system.

2.3 Existing Probe Technologies

There are existing technologies that either were intended for use as probes or which can be easily put into use as probes although their intended use was something else.

Many of the technologies presented in the following section were developed as component adaptation techniques. At first glance adding monitoring code to a component appears to be a simple component adaptation; however the difference is in when the adaptation occurs. Component adaptation has largely been presented as a mechanism to overcome component mismatch [18, 19, 22, 15]. The mismatch can either be interface mismatch or behavioral (functional) mismatch [18, 15]. The issues that component adaptation attempt to address are largely integration or design time issues. They seek to allow the integration of adapted third-party components into software systems. In this thesis we present probing in terms of constructing a generic monitoring infrastructure for use with already integrated systems. This difference, while significant, does not completely undermine the utility of the previously developed adaptation techniques, as we will see in the following sections. Considering post-integration adaptation does however cause some of the characteristics of the adaptation mechanisms to be seen in a new light. In the sections that follow we will present mechanisms that can be useful for probing. Specifically, we will examine their use in a post-integration instrumentation role.

2.3.1 Open Source

If the source code for the component to be monitored is available, we can directly insert the monitoring code. This approach offers tremendous flexibility and enables the monitoring of any facet of a component's state or behavior. However, this flexibility comes at a cost, because monitoring code is intermingled with application code. This is undesirable because the two types of code for different purposes are intermixed thereby reducing cohesion. The different types of code will likely undergo separate evolutions, which will be made more difficult by the lack of separation. The individual responsible for inserting the monitoring code will most likely be responsible for all future maintenance costs as the initial component developer can claim any undesirable behavior is introduced by the monitoring code. Additionally the insertion of monitoring code is not assisted or guided by any tool. The result is that the monitoring code is often inserted ad hoc. This can complicate both the insertion process and the subsequent removal of the monitoring code.

2.3.2 Wrappers

Wrappers are commonly used for three purposes. One, they are used to adapt the interface of a component to allow for its integration into a system that expects or requires a different interface. Two, they provide a mechanism for adding functionality to a component. Three, wrappers can be used to wrap the interface of several components so that the system outside the wrapper refers to the components through a common interface. These uses have been formalized as the adapter, decorator and facade design patterns respectively [15].

Since we are considering the insertion of probes after integration, the interfaces in the system should already match so the probes should be as non-disruptive as possible. The decorator pattern [15], which is used to add functionality, is the most useful form of wrapping in terms of probing. A decorator is inserted in the control flow of a program between the method call and the actual method invocation. This provides access to the phase immediately before and after the wrapped methods invocation. During these before and after phases the wrapper can collect information, perform any desired processing and send the information to the monitoring system. Wrappers monitor a component by intercepting control at a component's port but outside the component boundary

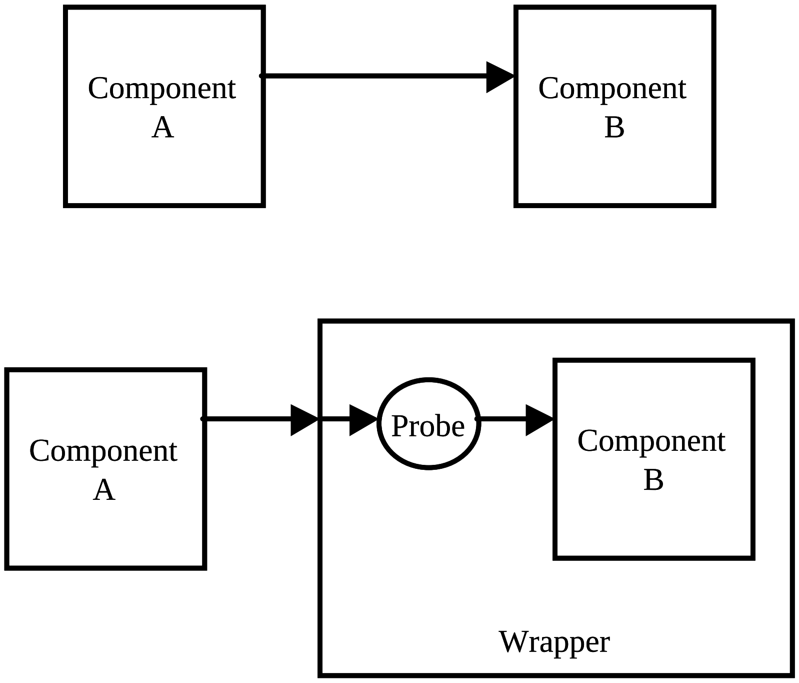


Figure 2.2: Inserting a probe with wrappers.

The use of wrapping for probing has several advantages. One advantage is wrappers keep the probe code separate from the component code. A second advantage is wrappers can be built to intercept all methods invoked on their target object including methods inherited from a super-type.

Wrappers also have several disadvantages. One, the system must be changed to refer to the wrapper instead of the original object. This could require significant changes to the system. Two, if the wrapped object invokes methods on itself they will not be monitored, since the control will not pass through the decorator. Three, wrapper code is highly coupled to the object it wraps and is thus not likely reusable. This last problem would be mitigated by the possibility of automatic code generation, also the probe code used in the wrapper could be quite general allowing for its reuse.

Many authors have written that wrappers provide homogeneity (transparency) [15, 18]. Homogeneity means that the because the wrapper conforms to the same interface as the original component, the system at large interacts with the wrapper as it would have with the original component. When wrappers are used at component integration time their use is indeed homogeneous to the system, since the system will be built specifically to interact with them. However, for the purposes of a generic probe infrastructure, probe insertion is assumed to at least be possible after integration. In this situation, code that referred to the original component must now be changed to refer instead to the wrapper. This necessity is alleviated if the capability to simply rename or change the package of the original component exists. However, renaming may not be possible if there is no source and repackaging may not be possible if there exist complex intra-package interactions.

The introduction of wrappers into a system post-integration requires the modification of the source code of components that refer to the object being wrapped. This is a disadvantage in terms of localizing the changes required to instrument a target system. However, it provides an opportunity to modify the interface of the target component to require information that would be useful to the probe code. An excellent example of this would be adding a new parameter that identifies the source of the invocation. Wrappers provide one of the few mechanisms that can successfully collect this information, which can be useful in reconstructing causal relationships from probed data.

Since, a wrapper has access to the interface of the component it is wrapping (and a reference to it in OO systems) the wrapper can perform some polling to determine the state of the component. This information can be included with the events it generates due to regular probe invocations or could even serve to guide the generation of events. In this later case, events would only be generated if the value of a certain property of the component was in a specified range.

2.3.3 Instrumented Connectors

Instrumented Connectors are the only probing approach presented here that specifically targets the communication mechanism of the software system being monitored. Instrumented connectors were developed to monitor inter-component communication. Specifically they were intended to monitor the architectural behavior of legacy systems and to aide in COTS component integration [4].

The instrumented connector API provides two ways to intercept communication in connectors. The first, externally instrumented connectors, are platform independent and can be used only with connectors that allow indirection, such as socket calls or Remote Procedure Calls (RPCs). An externally instrumented connector replaces a single connector in the target system with two connectors and a monitoring component (see Figure 2.3). Externally instrumented connectors are essentially a form of wrapper, in the mediator sense, that is specialized to wrap connectors.

Externally instrumented connectors are similar to wrappers, and the benefits and disadvantages are also similar. First, since they probe inter-component communication, they have access to events in a system at a fairly high level. This makes understanding the system and how to probe it simpler than mechanisms that probe lower level parts of the code. Second, externally instrumented connectors are conceptually platform independent. Any software system using connectors that support indirection can be probed with externally instrumented connectors. However, software support must be developed for

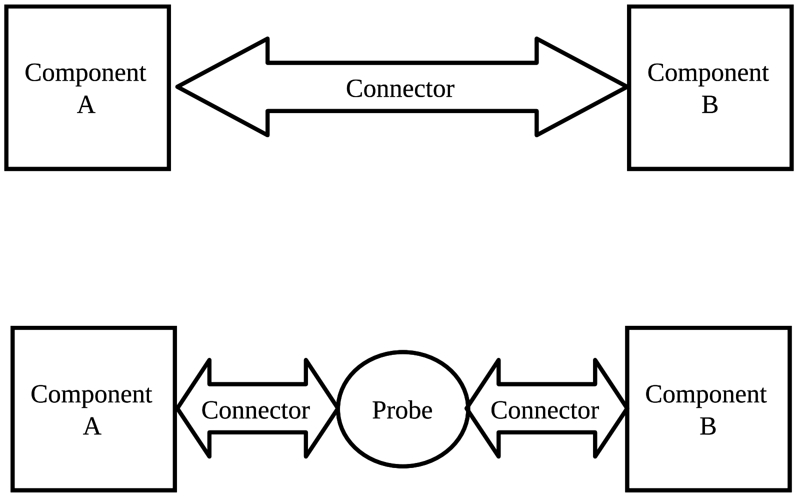


Figure 2.3: Externally instrumented connectors use indirection to insert a monitoring component into the communication flow.

the various connector implementations on a particular platform. Third, probe code is kept separate from implementation code. While it is often necessary to modify the target application to provide the indirection, this change is likely to be simple and require changes to very small portion of the code. Specifically, the implementation of the connector being mediated need not be modified at all. Externally instrumented connectors have the disadvantage that they are only possible for connectors that support indirection.

Modern operating systems rely on shared libraries and also provide support for the deployment of third-party software in the form of shared libraries. This has given rise to a vast quantity of software which utilizes this mechanism. Internally instrumented connectors, the second type of instrumented connector, are designed to take advantage of this. They can probe connections that use static, dynamic or shared system libraries [4]. To probe either static or dynamic libraries, a wrapper is constructed for the original library. The wrapper implements the same interface as the original library, but it now mediates the connection (see Figure 2.4). Static libraries are linked to a program at compile or link time. So the decision to enable instrumented connectors would necessarily be made by the component developer. Shared or dynamic libraries, on the other hand, are linked to the application when it is loaded or when it is needed. This dynamism requires that applications follow compiler-independent, operating system defined standards for the location of libraries and the creation of the links. These standards make possible intervention in the linking procedure and the retargeting of applications to utilize mediated libraries [4]. When a library method is invoked the mediated libraries can conditionally call the original library, alter the calls parameter values, return a different result (or exception) or simply execute probe code and transparently call the original library [4].

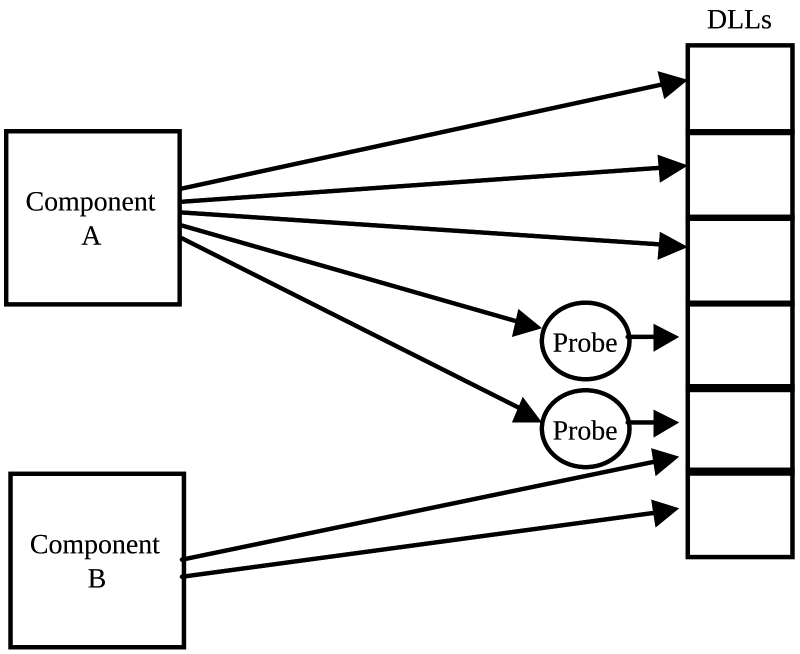


Figure 2.4: Internally Instrumented connectors can transparently function as probes by retargeting library calls to a wrapper that monitors their activities. Component A utilizes two instrumented libraries. Component B can continue to access the un-instrumented libraries.

The abundance of software which relies on dynamic or shared libraries leads to a significant advantage to the use of internally instrumented connectors as probes. Namely a portion of the necessary infrastructure to support the mechanism has already been developed and is in wide use. Instrumenting a shared library gains benefits beyond the ability to monitor a single target application. The same mediated library can be utilized with other applications.

Additionally, internally instrumented connectors can be applied to the operating system's libraries as well as third-party shared libraries. Relative to the number of software applications available, the number of shared libraries that comprise the operating system is small and yet monitoring them would provide access to many activities of interest in the target systems.

Source, while most likely not available, is not needed to implement an internally instrumented connector. In order to develop a mediating connector, only the interface of the original library is required.

It can be assumed that the target software system implements some useful functionality on its own. Where this functionality does not rely on shared libraries, it will not be possible to monitor it with instrumented connectors.

Mediated libraries are implemented dynamically within the address space of any process that is ordered to use them [4]. Within a process there are multiple scopes between which it would be useful for a mediated library to distinguish. The mediator could monitor all calls or it could monitor only those calls from specific threads, modules, libraries or users. Depending on the interface of the particular method/function/service being provided by the library it might not be possible to distinguish between different parties within each of these scopes, let alone limit the mediation to certain ones. Since instrumented connectors provide the capability to modify interfaces, it would be possible to modify the calling components to include identification information with the normal parameters to the library call. However, this need to modify source code from the system being monitored undermines a significant advantage of this mechanism.

The concept of creating mediated libraries is applicable for many modern operating systems. However, operating systems vary in the way they handle linking, loading and inter-process interaction. Therefore the implementations will be highly system dependent.

2.3.4 eJava

eJava is an extension of Java that was created to simplify the process of understanding and debugging multi-threaded systems [28]. eJava Programs are legal Java programs that generate events from which a computation can be constructed. A computation is a collection of events ordered by both time and causality. Causal relations define an ordering of events based on control flow, data flow and synchronization points [28].

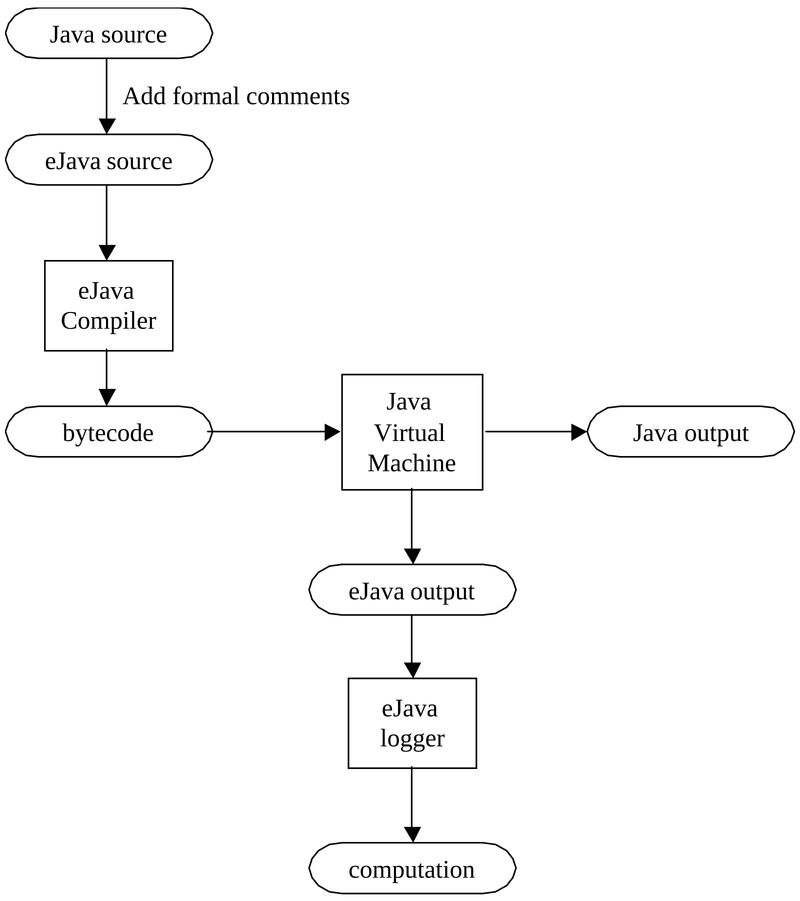


Figure 2.5: The eJava compilation process.

eJava provides a compiler which takes Java source code and outputs instrumented Java classes (see Figure 2.5). When the instrumented classes are executed, the normal output from the un-instrumented code is created. Additionally, eJava events are written to a file.

Events are generated in two ways in an eJava program. First, implicit events [28] are generated when:

- objects are created or finalized,
- methods are called or return,
- threads interact or are created or run,
- variables are read from or written to.

Second, explicit events can be generated at any point in the Java program where a Java statement can legally appear. They are generated by inserting `//+ perform` formal comments into the source code [28]. Explicit events can be used to provide additional information about the value of variables, or activities not logged by the implicit events.

eJava events contain a name, an associated thread and a time stamp. The events are written to a file. A logger (see Figure 2.5) uses this file to determine the temporal and causal relationships. The time stamps included in the events allow for a complete reconstruction of the temporal ordering. The causal ordering, however, must be reconstructed from the events and a set of connection rules defining causal relations for thread interactions. The logger outputs a computation that can be used as input to several tools such as partial order viewers and animators [28].

eJava is not intended to emit events to a general purpose monitoring infrastructure. However, many of the events it logs would clearly be of interest in a monitoring system, as would the computations that are currently created post-mortem. It would be necessary to extend the Logger to emit the events at run-time. Additionally eJava can currently only create computations for single JVM programs. Additional implicit event types and connection rules would be necessary to allow the creation of computations for distributed systems.

The primary benefit of eJava is that it allows the identification of thread interaction and thus provides a mechanism to understand complex synchronization behaviors. Additionally, all source code changes are accomplished in formal comments that are ignored by standard Java compilers, so reverting to UN-instrumented code is simple.

2.3.5 Active Interface

Active Interfaces were devised to support the creation of adaptable software components [18]. A component built with an active interface essentially supports two separate interfaces. The first is the component's own interface, the second allows for the adaptation of the component. An application builder can use the second interface to associate callbacks with the *before* and/or *after* phases of the component's methods. The active interface allows the callback to determine whether to *augment*, *override* or *deny* the method's activity. Active Interfaces provides a callback chaining mechanism that allows multiple callbacks to be associated with the a callback hook.

The individual wishing to monitor the system can associate callbacks with *before* and *after* phases of a component's methods. The callback code would contain the code to gather information and send it to the monitoring architecture. It would then return a value indicating it would augment the method's intended behavior, so that the original component code is executed.

In an active interface enabled component, the probe method is passed a `Context` object. The `Context` object encapsulates the method call that has resulted in the callback being executed. In addition to the method name and parameters it includes an object reference to the object on which the call was made. The consequence is that the probe code can collect other information directly by accessing the public interface of the object.

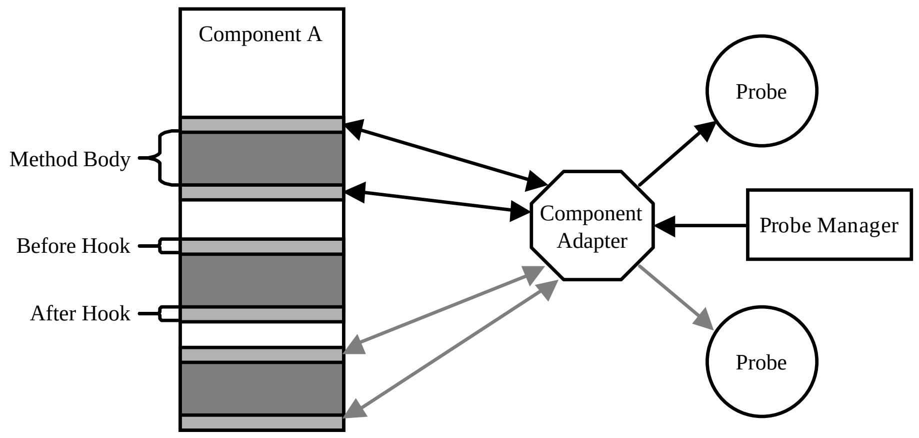


Figure 2.6: Active Interfaces can be used for monitoring by associating callbacks to probes with the hooks installed in a component by the AIDE compiler. The ComponentAdapter can independently register callbacks to different probes or to different methods within a probe.

This additional information can be included in the events that are generated.

The Active Interface Development Environment (AIDE) has been created to enable the use of Active Interfaces to adapt Java classes. The AIDE compiler takes the source of a Java class and inserts hooks in the before and after phase of each method. It also adds code to the class in order to implement the Adaptable interface. While the compiler is parsing the file it builds an instrumentation information file. This file documents each place where hooks were inserted in the class file and can be used to guide the subsequent probing of the target class. The AIDE compiler reduces the instrumentation of Java code to a change in the compilation process (see Figure 2.7). Traditionally, changes are made to the source code and it is compiled. If the source code is syntactically correct a Class file is generated. If not the developer corrects any errors and repeats the process. AIDE is seen as an additional step in the compilation process. When changes are made the source code is compiled with a standard Java compiler. If changes are necessary they are made. When the source code is syntactically correct, the AIDE compiler is run on it to generate instrumented code.

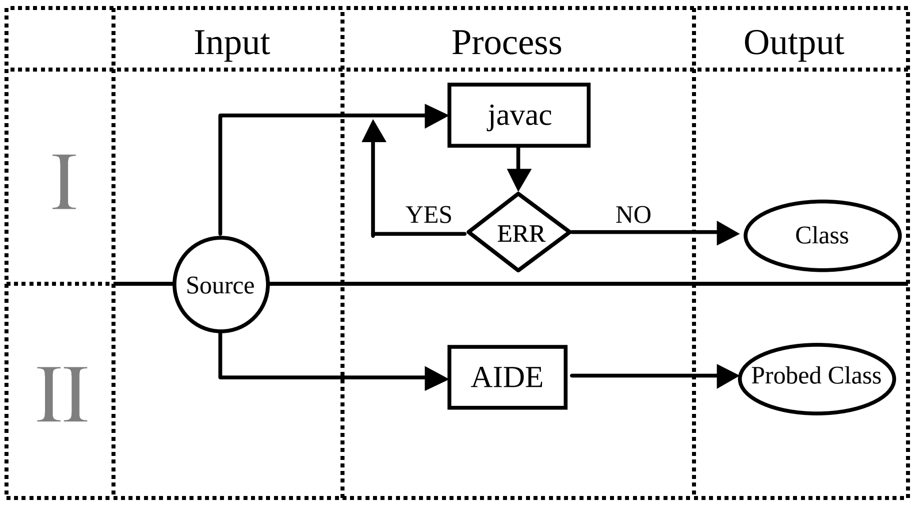


Figure 2.7: AIDE Compilation Process. AIDE compilation is a second step in the compilation process. The AIDE compiler is run on the source code after the source compiles without error.

Active Interfaces provide benefits for probing. First, the probe code is kept separate from the component code, this allows for easy separate evolution and/or removal of the monitoring code. Second, while currently a tool-set (AIDE) exists only for Java, active interfaces is not conceptually restricted to use in object oriented languages. Third, the system outside the component refers directly to the original component. All of the probing takes place hidden behind the component's interface. This reduces the impact on the system by localizing the changes required. Fourth, Active Interfaces can be used to probe all of the methods of an object. This includes methods that would not be available through inheritance due to access modifiers and methods that would not be available to wrappers because they are called from inside the component. Fifth, with Active Interfaces the association of callbacks with component hooks is achieved programmatically at run-time. The benefit of this late-binding is that Active Interfaces readily support the addition or removal of probes at run-time.

The primary disadvantage of Active Interfaces is that with current tool support source code is required for the insertion of hooks or the component developer must have already put hooks in place.

2.3.6 Inheritance

Inheritance provides functional and structural reuse and allows object oriented systems to model the real world. This capability is realized the ability to extend or specialize a class. Method overriding (or redefining) allows subclasses to define a method with a signature matching that of their superclass. When the newly defined method is called on an object of that subclass, it is executed instead of the overridden method in the superclass. For components written in object oriented languages, inheritance can be used to extend the class hierarchy and method overriding can insert monitoring code into the subclasses. To probe a class in this way, you would need to create a subclass that overrides each accessible method of its superclass. Each of these new methods could execute probe code before and/or after executing the original method in the superclass.

Inheritance has several desirable qualities that make it a good approach for probing. First, in contrast to wrapping, inheritance allows the probing of methods even when the invocation originates inside the object being probed. Second, the source code for the class that is actually being instrumented is not needed. This makes inheritance a good candidate for probing classes where no source code is available. Third, probe code is kept separate from the code being monitored allowing for easy separate evolution.

There are several drawbacks to using inheritance to insert probes. One drawback is that many classes may need to be subclassed to probe a component or system. Subclassing many classes in an existing hierarchy can lead to a complicated class hierarchy that is difficult to further extend. Two approaches are shown in Figure 2.8. In the first example, the existing hierarchy is preserved. The result being that when B is extended to add probes, all of the methods of A must be overridden in B’ since A’ is not a parent of B. The second example requires changing the code of B to extend A’ instead of A and therefore

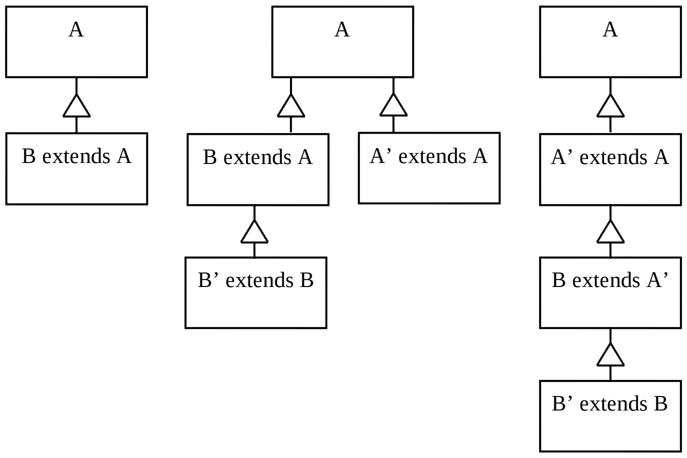


Figure 2.8: The Downside of probing using inheritance is that the object hierarchy becomes cluttered. In order to probe A and B one of the two hierarchies on the right is necessary. (Objects A' and B' are probed version of objects A and B.)

incurs the drawback of source code modification. A second drawback is that in object oriented languages with access modifiers inheritance may not be able to provide probing for all of the methods of the target system. For example in Java, a private method may not be overridden and therefore could not be probed using inheritance. A third drawback is that, while access to the source code for the class being extended is not needed, the system must be changed to create objects of the subclass instead of the superclass. This is detrimental in post-integration probing as it could result in widespread changes for frequently used classes. If the original component was created by a factory, the change would be localized to the factory code [15]. However, in the general case, this would require that the individuals probing the system have access to its source code of any component that creates objects of the subtyped class. While the necessary changes could be widespread, they would be fairly easy to generate automatically. Also, if the source code for the class being monitored was not available, but that for the system in which it is used is, the benefit of being able to probe the class in the absence of its source code might outweigh the complications

2.3.7 Binary Code Instrumentation

A number of tools have been developed to monitor software by instrumenting program binaries. Basic block counting tools (Pixie [32], Epoxie [36] and QPT [23] count the number of times that each basic block is executed. These allow developers to identify critical sections of code and bottlenecks. Pixie [32], QPT [23], MPTRACE [13], ATUM [1] generate address traces. These tools were all designed to fulfill a certain purpose and are not easily adapted for use in a general monitoring system. In the remainder of this section we present two systems that take a more general approach to instrumenting programs without altering source code.

ATOM

Analysis Tools with Object Code Modifier (ATOM) was designed as a system for building customized program analysis tools [34]. ATOM has been used to build a variety of tools including: basic block counters, profilers, memory recorders, instruction and cache simulators and branch predictors [34]. ATOM uses the generic Object Code Modifier (OM) to give the tool producer a high-level view of a program's object code and provides infrastructure to support its manipulation [34, 33]. ATOM can be used to insert calls to analysis routines into a target application at link-time. The system instrumenter can choose what analysis routine to call, where to call it from, and what arguments to pass [34]. It is these capabilities that make it useful to a system instrumenter for the collection of general information from running systems.

ATOM requires three files to instrument a target module: the target object module, a file containing an instrumentation routine, and a file containing analysis routines. The instrumentation routine uses ATOM's high-level view of the object module to determine where to insert calls to analysis methods and what information should be passed to them as parameters. The analysis routine file contains all of the data and procedures that are

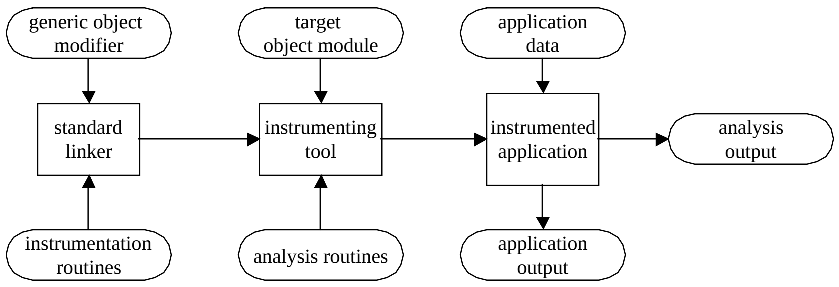


Figure 2.9: The ATOM tool-building process.

called from the instrumented target file (and any additional helper procedures) [34].

Instrumenting a target module with ATOM is a 2 step process (see Figure 2.9). First, a standard linker is used to link the instrumentation routines with the generic object code modifier (OM) [33]. This step results in a custom instrumentation tool. Second, the custom tool is applied to the target object module generating an instrumented executable.

The use of ATOM for probing has a number of benefits. ATOM works on object modules and is therefore applicable to a wide range of languages and compilers. It has been tested with FORTRAN, C and C++ [34]. Since ATOM transforms object modules, no source code is required. Furthermore, the object modules of the target system are preserved. All of the changes to are kept in separate transformation files. ATOM is very flexible, it allows the insertion of calls to monitoring routines before or after any program procedure, basic block or instruction. Fourth target system source code is not required. Finally, ATOM is designed so that it provides the analysis routines with data and text addresses as if the application were running uninstrumented. This is very valuable for the detailed analysis for which atom has been used.

The primary downside to ATOM is that it is currently implemented only for the DEC Alpha AXP under OSF/1. In addition to this it only works with non-shared libraries.

2.3.8 Java Bytecode Modification

The Java Virtual Machine (JVM) specification is designed to support the execution of the Java language. The JVM executed bytecodes are essentially equivalent to the binaries executed by other machines [26]. Java source code files are compiled to class files which contain the methods and data of the class in physical machine independent bytecodes. These class files retain a great deal of symbolic information. A number of techniques have been developed which utilize this information and the structure of methods within the class file to modify Java applications by modifying their bytecodes [25, 21, 8, 10]. These techniques have been put forth as offering possible solutions to a host of problems including: extending the Java environment, integrating classes with specialized environments, implementing aspect oriented techniques, adding functionality, adapting the behavior or interfaces of a component, providing load-time reflection, or providing information for run-time analysis tools.

These technologies are significantly similar so we do not discuss them each in a separate section. Instead, we briefly present each and then present the similarities and differences in the approaches. We then discuss the merits of using this approach to probe Java code.

Each of these mechanisms uses one or both of two approaches to modifying bytecodes. The first approach, producing modified class files for later use, is the primary mechanism used by BIT [25] and the JavaClass API [10]. The second approach, extension of the Java class loader to apply deltas or transformations to classes as they are loaded, is supported by Binary Component Adaptation (BCA) [21] and the Java Object Instrumentation Environment (JOIE) [8]. Both approaches are conceptually possible with all of the mechanisms but tool support for load time transformation is currently only provided for JOIE and BCA. A significant downside to the load-time transformation mechanisms that are provided is that the implementations are dependent on the version of Java supported by the JVM in question and the particular JVM implementation. BIT and the JavaClass API by requiring the user to create modified class loaders themselves are not dependent on a specific version. At the end of this section we present an alternative implementation of load-time transformation that is not version-specific.

Binary Component Adaptation

Binary Component Adaptation (BCA) was developed to promote the easy adaptation and evolution of software components [21]. It focuses on how to combine independently developed components into a working system. BCA works by applying delta files to a class file. Delta specifications contain information about what changes must be performed to which class files at load time. They also include the source code for any new methods that will be added to the class. A delta file compiler is used to translate delta specifications into a compact representation that includes what classes to change, the changes to be made, and possibly any bytecode required for new methods. At load time, the modifier component gets the target class file from the class loader and applies the delta file to it. This modified class is then passed to the JVM's verifier that checks that it does not violate any JVM rules. Since verification occurs after class file modification, the adaption has no effect on security [21].

BCA currently supports a fixed set of class file modifications. It can be used to: rename classes, change super classes, add or change interfaces, add or rename methods, and add or rename fields. It does not allow the addition of bytecodes to existing methods. However, through composition of the allowed operations, you can achieve probing before and after method calls, essentially creating an inner wrapper. This could be achieved by renaming the method you wish to probe and adding a new method that has the exact signature of the original method. The new method would call a monitoring component and then call the original method. Likewise after the return from the original method, the

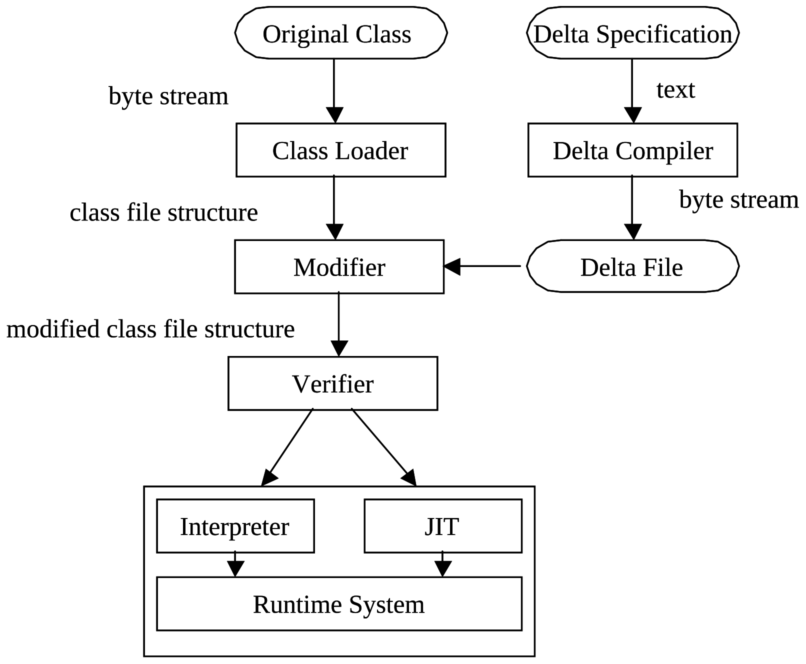


Figure 2.10: Binary Component Adaptation (BCA)

new method would again call the monitoring method prior to returning the value from the original method to the caller.

In [21] the authors suggest that if the complete set of classes can be determined statically the adaptation of the target class files could be done off-line to overcome the (reasonable) slow downs that result during class loading. We feel that the creation of probed code as described in the previous paragraph could proceed in the absence of the complete set of classes without compromising the system.

The JavaClass API

The JavaClass API provides a general purpose framework for creation, or transformation of Java bytecode [10]. It has two intended uses, the first is static analyses of class files in the absence of source code. In this application bytecodes are not modified. The second use is “generic” and allows the JavaClass APIs to be used to modify class files off-line, modify them at load-time or even create entirely new classes at run-time. It is load-time modification that appeals to the system instrumenter.

The generic API requires the development of a transformer class. The transformer class applies the desired changes to a series of files. It does this by creating a `JavaClass` from the loaded class and then modifying the bytecode using the JavaClass APIs. It would be possible to build this capability into a class loader and have the transformer automatically run on all or selected classes as they are loaded.

The JavaClass API models every element of a Java class file as an object and provides APIs to manipulate them [10]. The ability to create entire classes at run-time gives some idea of the flexibility of this mechanism. In fact it is possible to modify the target class file in any way that results in valid (as defined by the JVM) bytecode.

The most interesting application for a system instrumenter is the insertion of calls to a monitoring method, so we will briefly discuss the addition of instructions. During modification of the class file symbolic references are used to refer from one instruction to the next, only during finalization are the references converted to the concrete references required for bytecodes [10]. This greatly simplifies the addition of new instructions. The transformer uses calls to the JavaClass API to insert new bytecodes at the predefined locations.

The down side of the JavaClass API is the need for the program transformer to add instructions as a sequence of bytecodes. The system instrumenter must either be able to write the bytecode directly or extract the necessary bytecodes from a compiled class.

Bytecode Instrumenting Tool

The Bytecode Instrumenting Tool (BIT) was designed to allow the instrumentation of bytecode to provide insights into the dynamic behavior of the systems [25]. BIT is a set of classes that allow the user to insert calls to analysis methods before or after methods, basic blocks and individual bytecode execution instructions [25].

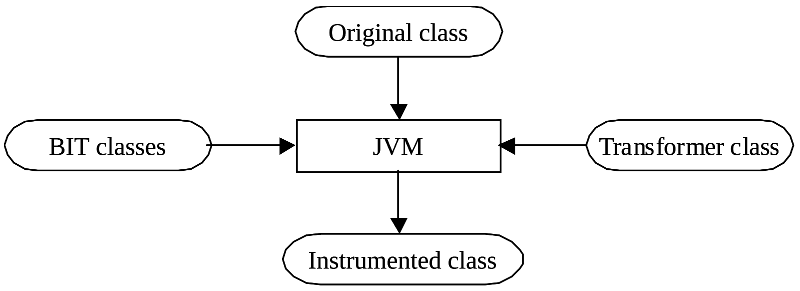


Figure 2.11: The Bytecode Instrumentation Tool is a set of classes that allow a user to create a transformer to modify Java classes. The result is a modified class file that can run on a standard JVM.

BIT is designed for the off-line generation of class files (see Figure 2.11), however much like the previously presented mechanisms, load-time support could be built into a class loader. There are three issues that will limit the utility of BIT in the construction of a general probe infrastructure. First, the current implementation only supports the invocation of static analysis methods using the `invokestatic` bytecode (The author states that this limitation implies that no objects may be associated with the method) [25]. Second, BIT restricts the number of arguments to the analysis method to one object. Future support for a greater number of arguments is intended [24]. Third and most significant, when adding method calls BIT changes the code buffer, the current implementation does not correct the references to exception handlers in the modified buffer. As a result incorrect exception handlers could be invoked leading to run-time errors [25].

Java Object Instrumentation Environment

The Java Object Instrumentation Environment (JOIE) was designed as a toolkit for the load time transformation of Java classes [8]. In JOIE transformations are presented as mechanisms to implement behaviors that are orthogonal to the purpose of the Java class they transform [8]. The instrumentation of code from monitoring purposes is just such an orthogonal behavior.

Like the JavaClass API, JOIE utilizes a high level object view of Java class bytecodes. In JOIE, `ClassInfo` objects are created from the bytecodes of a class. `ClassInfo` simplifies the modification of a target class and correctly updates exception handlers [8]. `ClassInfo` allows transformers to:

- set or unset modifiers;
- add, remove, or rename fields or methods;
- change method signatures or field types;
- adjust the list of interfaces implemented by the class;
- adjust references to fields or methods to point to new fields or methods;
- adjust the value of embedded constants;
- manipulate the inheritance hierarchy [8].

JOIE comes with a modified class loader that invokes each registered transformer on each class (or some subset there of) as it is loaded. To help protect against malicious code the current implementation does not allow the insertion of new transformers once the first class is loaded [8]. The JOIE APIs allow for load time reflection that transformers can use to guide the transformation process.

One way to implement probes with JOIE would be the creation of inner wrappers as described in the BCA section above. A target method would be renamed. A new method with the exact signature of the target method is inserted. This new method first calls out to probe code and then invokes the original method.

Comparison of Byte Modification tools

These solutions have considerable common ground in they way they approach the adaptation of class files. All operated directly on the bytecodes of the class files directly thus do not require source code. All codify the changes to be made to a class external to the class itself. This makes it easy to modify the application or remove the transformers.

JOIE, BCA and the JavaClass API have the capability to disrupt homogeneity, that is, they can change the way external components view the object. This is a measure of their flexibility, but not really a desirable capability in terms of probing. A developer using any of these techniques to probe a class must take care to preserve homogeneity.

In JOIE, the JavaClass API and BIT transformer classes are constructed to programmatically alter the class files. JOIE and the JavaClass API allow almost any change to be made to a class. They provide a high-level view of class structures that allows creators of transformers to view the class file as a collection of objects. However, if a transformer wishes to actually insert new code it must be inserted directly as bytecode. In contrast BCA and BIT allow less freedom in the type of change that can be accomplished but they do not require the user resort to bytecode. In BCA delta specifications are written with Java-like syntax and in fact new methods to be added are written directly in Java. After compilation the delta file is applied to the target class. Bytecode for the Java method to be inserted is automatically extracted from the delta file and inserted into the class. Bit allows only the insertion of method calls and handles this without requiring the transformer designer to resort to bytecode.

JOIE, BIT and the JavaClass API can be used to create transformers that are widely applicable to many class files. This is because transformer authors can perform tasks relative to class structures, and perform reflection like tasks. BCA's ability to do the same is limited by the higher level at which the delta files view the class. Delta files deal with quantities in terms of their names in the class. It does not provide a mechanism for discovering this information and by the very nature of load time transformation, Java's reflection facility is unavailable. The result is that delta files must be tailored to the class files they are designed to modify.

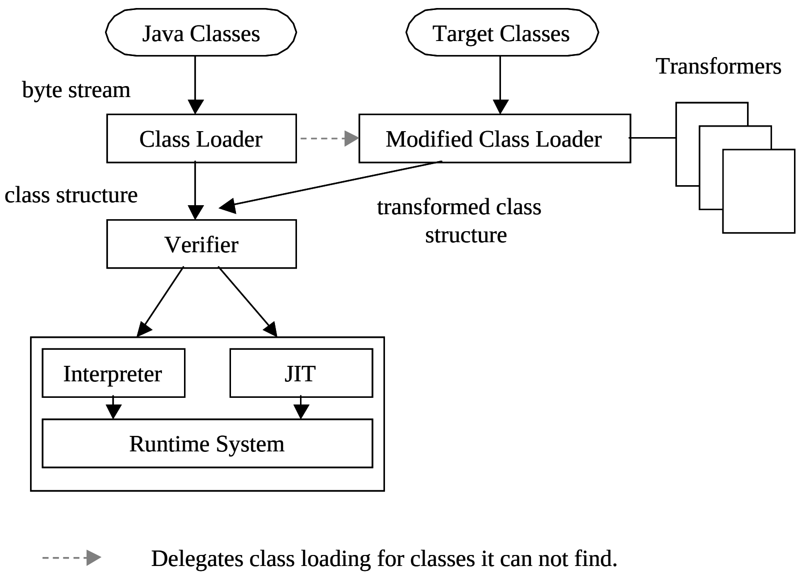


Figure 2.12: Modified class loading scheme that would allow the load-time transformations of class files by any of the mechanisms described in this section. This scheme requires no modifications to the JVM.

All of the above technologies could be used (with some limitation as to target classes) at load-time without requiring modification to the System Class Loader. In Java 1.2 and higher, Class Loading uses a delegation scheme whereby if the developer so chooses they can create a modified class loader with their own behaviors added. They can specify this class loader be used explicitly in the `Class.forName( className, classLoader, initializeBoolean )` method. This alone is not entirely useful as it is desirable to apply the transformations or delta files to classes that are created through the normal means in the body of third-party code.

The utility of delegation comes in here, if the class from which the new class is being constructed was loaded through the modified class loader and the system class loader cannot locate the new class, the modified class loader will be used to instantiate it through delegation. In order to force the classes to be loaded through the modified class loader three things would be required. First, no class which the instrumenter wishes to transform must be available in the CLASSPATH. Second the modified class loader must know where to find all of the files that should be modified. Third, the target application's main class must be explicitly loaded using the modified class loader. The simplest way to achieve this would be to launch the application from a generic shell that specifies the class loader. This method has the benefit that it is not tied to any one implementation of the JVM. As long as Java 1.2 style class loader delegation continues to be supported this mechanism would work. It has the limitation that the java.* packages would not be accessible for instrumentation since they would still be loaded through the system class loader.

2.3.9 API Polling Probes

API polling probes collect information from the target system by polling the public interfaces of its components. API polling probes are distinct from open source probing because with an API polling probe, no changes are made to the source code of any of the components that comprise the system. These probes can be written in any programming language that allows the probe to interact with the component's APIs. Polling probes are triggered by events external to the target system. Polling can be performed on a schedule or to fulfill specific information requests. Polling probes are especially useful in any system that presents public APIs that expose interesting aspects of the systems overall state. In addition to collecting information about the target system, polling probes can be used to monitor the environment in which the target system is executing. For example if the operating system in which the system is executing provides APIs to interact with the file system, polling probes could collect information about disk usage and free space. The primary limitation of API polling probes is that they only have access to information that is exposed in the target component's API.

2.4 Categories of Probe Variation

From our examination of the above-mentioned probes a number of categories of variation have become evident. In this section we describe some ways in which the various probes differ and present a series of tables that summarize the characteristics of the various probes. This information forms a useful summary to guide the selection of probe mechanisms.

2.4.1 Triggering Mechanisms

As discussed in the chapter introduction probes monitor the communication and state of software system and the state of the environment in which the software runs. Probes emit events to the monitoring infrastructure, either when something occurs in the target system, on a schedule, or on request.

Probes that emit events entirely in response to activities in the target system are *passive* probes. Passive probes can be triggered by the invocation of methods in the target system, the activation of a mediated connector, or the execution of probe method calls that have been inserted directly into the source code or binaries of the target system. Passive probes help to capture the dynamic behavior of the target system.

Active probes are triggered by events that are external to the target system. Active probe events can be delivered to the monitoring system on a timed schedule or in response to specific requests for information made by the monitoring system. State in the target system, could be monitored by active probes that poll the public interfaces of components and generate events that encapsulate the state information. In addition to collecting information about the target system, active probes can be used to monitor the environment in which the target system is executing. This can include network loads and latency, disk usage and available space, or the liveness of remote systems with which the local system must interact.

Hybrid probes fall somewhere between active and passive. Hybrid probes may use passive probe technology to collect information from the target system but instead of constantly passing events on to the monitoring architecture, they wait for a schedule or information request. Hybrid probes are useful in situations where there are some activities in the target system that will only be of interest in certain circumstances or simply to reduce the load on the monitoring system. Hybrid probes can perform an initial filtering pass over the events generated in the target system, forwarding only events that meet certain criteria to the monitoring architecture. Alternatively, high frequency events in the target architecture could be aggregated and reported to the monitoring system periodically.

API polling probes are the only probe type presented in this chapter that serve in the active roll. API probes are never in the normal flow of control through the target system. Instead they execute in a separate thread and collect information on a schedule or in a response to requests from the monitoring system. All of the remaining probe mechanisms presented in this chapter collect information passively. The probes are inserted by various mechanisms into the control flow of the target system where they are triggered by events in the target system. All of the passive probe mechanisms are capable of acting as hybrid probes.

2.4.2 Probe Dependencies

In this section we discuss the dependencies of the various probe types. We distinguish between implementation dependency and conceptual dependency. Implementation dependencies indicate that the probe technology, has a dependency due to existing tool support. A conceptual dependency (Table 2.2) exists if the probe technology is dependent on something without which it would be unable to exist.

Table 2.1 shows the implementation dependencies of the probes presented in this chapter. Table 2.1 shows only those probes with dependencies which can be overcome by an extension of the existing toolsets. Internally instrumented connectors have currently only been implemented in Windows NT. They could however be implemented in any operating system that supports shared libraries. eJava tools only support java. The concept could be extended to any language that provided a means for identifying threads during execution. Active Interfaces support for languages other than Java could be built by creating parsers for the language that insert the Active Interface hooks. ATOM like systems could be built for other operating system. The primary requirement is that the target languages generate object modules that are linked to form executables.

Probe	**Dependency**
Internally Instrumented Connector	Windows NT
eJava	Java
Active Interfaces	Java
ATOM	DEC Alpha AXP under OSF/1

Table 2.1: Probe Implementation Dependencies.

Conceptual dependencies are stronger than implementation dependencies. A conceptual dependency can not be overcome by developing new tools. The following table summarizes the conceptual dependencies of the probe types presented in this chapter. The dependencies in the table are straightforward.

Probe	**Dependency**
Open Source	source code
Externally Instrumented Connector	connectors that allow indirection
Internally Instrumented Connector	shared libraries
Inheritance	Object Oriented Languages
Java Bytecode Modification	Java Bytecode
ATOM	Object Modules

Table 2.2: Probe Conceptual Dependencies.

2.4.3 Probe Location

The probes described in the previous section work to monitor the target system by capturing information at a number of different places relative to the architectural view of the target system. As described in the Models of Target Systems section, an architectural view of the system consists of components and connectors. We use this view to broadly define probes as being component boundary intrusive, connector intrusive or architecturally intrusive. Table 2.3 summarizes where the various probing techniques discussed in the first part of this chapter are located relative to the system architecture. It is important to note that while the table shows the primary location a mechanism would target, many of the mechanisms have the capability to cross over. For instance any of the component intrusive mechanisms could conceivably be utilized on actualized connectors as well.

A number of entries in the table deserve a little explanation. First, while internally instrumented connectors insert a new component between the component and the library, they are not considered to be architecturally intrusive. This is because the target system does not need to change to refer to the inserted component. The redirection is handled by the modified linking mechanism. Second, wrappers and inheritance while focusing primarily on the component are considered architecturally intrusive because the changes required to utilize them are not localized to the component being probed.

Location	**Probing Mechanism**
Component Intrusive	Active Interfaces Open Source eJava Binary Code Modification Java Bytecode Modification ATOM
Connector Intrusive	Internally instrumented connectors
Architecturally Intrusive	API Polling probes Wrappers Inheritance Externally Instrumented Connectors

Table 2.3: Architectural probe location by probe type. Probe Location is determined according to the mechanism's primary focus.

2.4.4 Probe Scope

Another defining characteristic of passively triggered probes is the ability to distinguish between the different possible entities that could have called the method which lead to the triggering of the probe. Can the probe technology determine what process, thread, module, user is responsible for triggering the probe invocation. This ability is useful because it allows selective targeting of probe code. Only code invocations caused by certain entities would result in an event being emitted into the monitoring architecture.

Probes that target components can distinguish between instances of the component while those that focus on the connectors are unlikely to be able to do so.

Any probe written in Java, through which execution is redirected at run-time can determine what thread it is executing in. If a system had a predefined set of threads that would be executing the probe could use thread identity as a condition for the delivery of events to the monitoring system. Alternately, the executing threads identity could simply be included in the event.

2.4.5 Probe Insertion Mechanisms

Table 2.4 summarizes the how probes are inserted for the various mechanisms.

Probe	**Insertion Mechanism**
Open Source	source code modification.
Externally Instrumented Connectors	connector indirection.
Internally Instrumented Connectors	library replacement and intervention in the linking procedure.
eJava	annotation of source code followed by compilation.
Active Interfaces	compiled in.
Inheritance	extension of the class hierarchy.
API Polling Probes	none.
ATOM	linked in.
Binary Component Adaptation	load-time transformation.
The JavaClass API	transformation at or before load-time.
Bytecode Instrumentation Tool	pre-load-time transformation.
Java Object Instrumentation Environment	load-time transformation.

Table 2.4: Probe Insertion Mechanisms

2.4.6 Black Box Probing

A probe mechanism is considered to take a black box view of the component if no knowledge of the internals is required [21]. Note that this is different from requiring source code. A probe mechanism that requires source code but does not require knowledge of implementation details is still considered black box.

Table 2.5 shows whether the probing mechanisms are black box techniques and whether they require source code. The probing mechanism is considered to require source code if it currently has an implementation dependency or a conceptual dependency on source code. Active Interfaces and eJava both require source code due to current tool support. However, eJava also allows the insertion of explicit calls to its event logger anywhere in the source code. For explicit event generation eJava has a conceptual depen-

Probe Mechanism	**Source Code**	**Black Box**
Open Source	yes	no
Wrappers	no	yes
Externally Instrumented Connectors	no	yes
Internally Instrumented Connectors	no	yes
eJava	yes	no
Active Interfaces	yes	yes
Inheritance	no	yes
API Polling Probes	no	yes
ATOM	no	yes
Binary Component Adaptation	no	yes
The JavaClass API	no	yes
Bytecode Instrumentation Tool	no	yes
Java Object Instrumentation Environment	no	yes

Table 2.5: Black Box Probes

dency on source code. Also if eJava is used to generate explicit events it loses its status as a black box probing mechanism. Similarly ATOM, the JavaClass API, the Bytecode Instrumenting Tool and the Java Object Instrumentation environment allow the system instrumenter to violate the blackbox conditions. If they are used to examine the internal implementation of the components they instrument they are no longer black box.

2.5 Conclusions

In this chapter we presented a number of probe mechanisms and described their properties. This description can serve as the basis for a system instrumenter selecting a probe mechanism. In the next chapter we present a high level design for a system that can make any of these probe mechanisms more useful in monitoring roles by providing a means to interact with them.

Chapter 3

Probe Run-Time Infrastructure

3.1 Introduction

Probes can be added to the system statically at design (compile) time. However, this severely limits the utility of the monitoring system. It will not be necessary or desirable to have output from all the possible probes that could be deployed in a system at all times. Even if the monitoring system simply ignored the majority of the probe output this setup would tax the communication and computation resources available to the system. Instead, the need for a Probe Run-Time Infrastructure has been identified. The probe run-time infrastructure standardizes the run-time deployment and control of probes and probe event delivery. By doing this, the monitoring system or individuals controlling it can determine what parts of the system to monitor dynamically. In this chapter we discuss the issues that such an infrastructure must tackle and present a high level design of a Probe Run-Time Infrastructure that is heavily influenced by [3]. In the next chapter we present an implementation of the high level design for the control of Active Interface probes.

3.2 High Level Design

A high level design for Probe Run-Time Infrastructure must address a number of issues. What actions will be possible at run-time. What should the protocol be to support these actions for a wide variety of probe types. How should probes and probe insertion points be identified in the system. What structure will probe events have and what meta information will be associated with all probe events. How will the system handle exception conditions arising during its attempt to support the desired interactions.

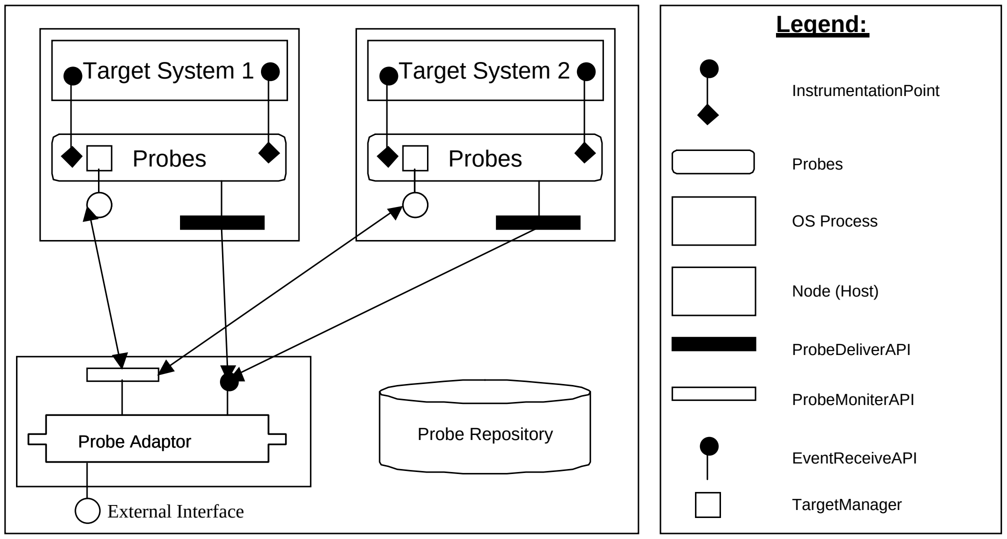


Figure 3.1: The Probe Run-Time Infrastructure.

One of the key elements in the Probe Run-Time Infrastructure is the Probe Adapter. The Probe Adapter is responsible for interacting with the monitoring system and the deployed probes. It provides a standardized interface to allow probes of different types to be controlled at run-time in a standardized way. In the rest of this section we present the elements of the high level design in respect to the Probe Adapter.

3.2.1 Probe Structure

We will discuss two granularities with respect to probes. The smaller of the two, the Probe, is an individual instance of some probe technology. Their scope will be somewhat dependent on the underlying technology on which they are based. The larger granularity is the *probe configuration*. Probe configurations can include multiple probes and are the level of discourse for the run-time infrastructure. For example the probe run-time infrastructure will deploy probe configurations. For simplicity, we assume that a probe configuration generally will include probes based on only one technology. The implementer of a run-time infrastructure can choose to support multiple types if they wish.

3.2.2 Event Structure and Meta Information

The output of a particular probe may be of interest to multiple remote event consumers. Likewise multiple event consumers may wish to deploy and control probes. For that reason asynchronous event communication using a publish/subscribe protocol has been selected for the dissemination of events from producers and consumers.

There are two distinct event types implied by the presence of event producers and event consumers. One event type is for the delivery of events which are sensed by probes in the system. Event consumers would subscribe to these sensed events. The other type of event is a probe control event, these would be created by event consumers to control and insert probes and are termed Infrastructure events here.

Sensed events encapsulate information from the target system at a particular instant. They can capture information about behaviors observed by passively triggered probes in the target system or information collected by active probes in response to a timer expiring or a request from the probe infrastructure. The specific information that they include will depend on the underlying probe technology and the nature of the probe code. However, all sensed events will include: probe configuration name, host name, system identifier, event type, event data and a timestamp. These attributes will allow the gauge infrastructure to correlate events from the target system.

Event consumers will subscribe to receive sensed events by defining the values for the above parameters which events of interest to them should contain. Typically consumers would want to narrow their subscriptions as much as possible to prevent the delivery of unneeded events.

Infrastructure events are used to control probes at run-time. They must contain information that allows them to reach the appropriate probe configurations as well as any information needed to achieve their control function. In order to get to the appropriate probe configuration each infrastructure event must at a minimum contain an identifier which marks the message as a infrastructure event, a host name, systemID and a probe configuration name. The Probe Adapter is responsible for subscribing to infrastructure events and delivering them to the probe configurations. When the system begins execution on particular host it is assigned a System ID. The Probe Adapter subscribes to all infrastructure events which are targeted at the current host. When the Probe Adapter receives an infrastructure event it converts the event into the protocol understood by the probe configuration and delivers it to the configuration named in the event.

3.2.3 Infrastructure Events

As described in the chapter introduction there are a number of needs that must be met by an infrastructure to adequately control probes at run-time. The following probe infrastructure events meet these needs and were suggested in [3].

Deploy (host, Target System, probe-configuration-name, probe-configuration-module)

Install (host, Target System, probe-configuration-name)

Activate (host, Target System, probe-configuration-name)

Deactivate (host, Target System, probe-configuration-name)

Uninstall (host, Target System, probe-configuration-name)

Undeploy (host, Target System, probe-configuration-name)

Query Sensed (host, Target System, probe-configuration-name)

Generate Sensed (host, Target System, probe-configuration-name, event-name1 ... event-nameN)

Focus (host, Target System, probe-configuration-name, parameter1 ... parameterN)

Trigger Active (host, Target System, probe-configuration-name, probe-name)

Several of these events merit a little discussion. First, the Deploy event does not include the probe configuration to be deployed; instead the probe-configuration-module parameter is a URL. The actual module will be fetched with a point to point protocol. Second, when a probe configuration receives a Query Sensed event it will build and subsequently publish a Generate Sensed event. The Generate Sensed event contains a list of events that can be generated by the probe configuration. Third, the Focus event provides a mechanism to alter the behavior of the targeted probe configuration in a implementation specific way. Each probe configuration can build in behavior to be controlled by focus events. Finally, the Trigger Active event provides a mechanism to allow the monitoring system to request information from an active probe.

We have introduced a number of changes to the protocol specified in [3]. First, although not shown in the above list, each of the events has a matching response event that includes information about whether or not the event has resulted in changes to the probe infrastructure. Second, we added the Trigger Active event to enable the probe infrastructure to request information from active probes. Third, the original protocol did not specify that the Deploy, Query Sensed, or Generate Sensed events would require a system execution identification. Each of these events was seen to be querying a static property of a probe configuration and thus would not require the executing instance to be defined. However, for any Java enabled probe types this assumption breaks down. For a probe adapter and the probe configurations it manages to interact with the target system in a straight forward (and computationally cheap) manner, we assume they are all executing in the same Java Virtual Machine. Additionally, the aim of developing the Probe Run-Time Infrastructure is to support dynamism. It is easy to conceive of a probe configuration, for any probe type, for which the events it generates would either not be known statically or would possibly change dynamically. In this case, two statically identical probe configurations could generate different sets of events at run-time. Including the Target System in all of these events allows this limitation to be overcome.

The ability of the receiver to act on the events described above will depend on the current state of the target probe configuration. A probe that has not been deployed can not be activated. Figure 3.2 is a state machine that describes the events for which an action is possible relative to the state of the targeted probe configuration.

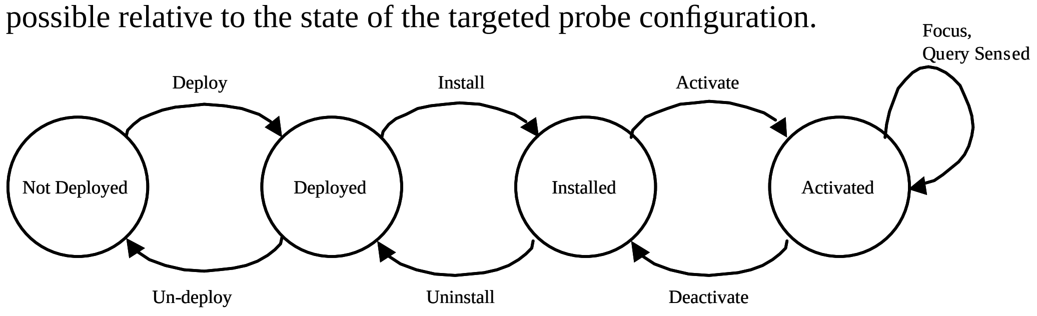


Figure 3.2: State machine figure showing the allowed state changes relative to the state of a particular probe configuration. All events for which no arc emanates from a particular state are not allowed and result in the generation of an error event and no state change.

3.3 Dynamic Probing

To allow the dynamic probing of a system at run-time it must be possible to deploy new probe configurations to a running system. Many of the probe technologies, presented in the previous chapter, do not directly support this need. Table 3.1 summarizes what would be required for each of the probe types to support dynamic deployment of probes. These requirements are over and above the requirements to probe the target objects with statically available probes. The necessity of a Probe Adapter for the various probe types is also assumed for each mechanism to support dynamic probing and provide integration with the monitoring system.

Open Source	Code must have been modified to accept new modules at run-time.
Wrappers	Wrappers that can except new code must already be place in the system.
Instrumented Connectors	Extensible library wrappers created and in place. The mechanism for interception of load/link activities in place.
eJava	Reworked logging facility in place to handle disseminate events at run time and to accept new configurations.
Active Interfaces	Hooks compiled in to target class. Target class must support self registration.
Inheritance	Class hierarchy extended, Sub classes must support dynamic addition of code.
ATOM	probe stubs linked into the application
Java Bytecode Solutions	The modified JVM or class loader must be used to run the target system. Probe code must probably be in place at the time of class loading
API Polling Probes	Can be deployed at any time as long as they can access the necessary object references to collect data.

Table 3.1: Preconditions for the run-time deployment of probes. This table assumes the presence of a Probe Run-Time Infrastructure for each probe type.

With the exception of Active Intefaces and API Polling probes all of the probe types must build in the ability to dynamically redirect their probing activities at run-time. The late binding of Active Interfaces automatically supports this. On the other hand a wrapper must be constructed with the ability to accept new code (conforming to some interface) and insert that new code into the flow of method calls between the wrapper interface and the interface of the underlying mechanism.

3.3.1 Handling Unsupported Interactions

It may not be possible for some probe types to support all of the above interactions. The Probe Run-Time Infrastructure for these probe types must still provide the interface defined here, but may ignore unsupported operations or return exception events when they are requested. For instance, if a particular probe implementation requires binding to the code at compile time, a deploy event is irrelevant.

3.4 Conclusions

In this chapter we have described at a high level the necessary elements for the creation of a successful probe run-time infrastructure. An implementation of this design for any of the probe mechanisms presented in chapter 2 would provide a flexible means for the collection of information from a target system. In the next chapter we describe an implementation of this design for Active Interface probing.

Chapter 4

Active Interface Probe Run-Time Infrastructure

4.1 Introduction

As mentioned in the Probe chapter Active Interfaces were developed to enable the adaptation of software components. In this section we first present several mechanisms that we have developed to facilitate the use of Active Interfaces to probe software components. We then present the design of the Active Interface Probe Run-Time Infrastructure. We describe several examples of probe types and uses that can be implemented with the Active Interface techniques. Finally we describe a demonstration in which the Active Interface Probe Run-Time Infrastructure is used to probe a simple client server system.

4.2 Facilitating Active Interface Probing

We have developed several mechanisms that make it easier and less intrusive on the target system to install probes into active interface enabled systems. Each Active Interface enabled component implements the `adapt.Adaptable` interface. As such it has an `adapt.ComponentAdapter` and accessor methods that allow other objects to get and set the `ComponentAdapter`. The Component Adapter provides a mechanism for the insertion of callbacks for the methods of the component. To add callbacks to a component, you must have access to an object reference for that component so that it can be assigned a `ComponentAdapter`.

We have developed a process of self registration through which each `Adaptable` component registers with a registration authority at the time of its construction. The registration authority creates a `ComponentAdapter` for the object and then has the option to insert callbacks. This process allows the standardization of the establishment of callbacks for Active Interface enabled components, and localizes changes to their constructors.

In addition to the self registration process, we have created an interface `AbleToBeProbed`. We will require components to implement this for use in the Active Interface Probe Run-Time Infrastructure. This interface provides the target class with a new attribute `componentName` and `setComponentName` and `getComponentName` methods. This allows for the possibility of naming individual components to align with an architectural specification. It also allows for the creation of probes which target only specific named objects of a class.

In Chapter 3 we described the Active Interface Development Environment (AIDE) Compiler. We created a modified version of the AIDE compiler that inserts the code necessary for self registration and implementing the AbleToBeProbed interface. The modified compiler performs these tasks conditionally. If it is passed the flag `-selfRegistration` followed by the exact lines of code that make up the self registration call. These lines will be inserted into the constructors of the target file. Likewise the compiler can be passed the `-ComponentName` flag followed by a component name. This name will be assigned as the initial `componentName` of the target class. With these changes the instrumentation of Java source code is completely automated. Once the new AIDE compiler is run on a source file it is ready for probing in the AIPRI.

In active interface probing callbacks are made from the target system to the monitoring system. It is highly undesirable to have errors anywhere in the probe code disrupt the execution of the monitoring system. We encourage probe developers to adopt as standard practice the catching of all exceptions that are thrown in the probe methods. Once caught an explanation of the exception should be encapsulated in an event and published to the monitoring system.

4.3 Design of the Active Interfaces Probe Run-Time Infrastructure

The Active Interface Probe Run-Time Infrastructure (AIPRI) is an implementation of the design specified in the previous chapter (Figure 4.1). It supports the run-time deployment and control of probes in a target system and the delivery of events generated by those probes to interested parties in the monitoring system. In this section we first present the content based event delivery system which we will utilize in our run-time infrastructure. We then discuss how the high-level protocol translates into Siena notifications. Finally we describe the design of the AIPRI.

4.3.1 Event Delivery: Siena

Siena is a event notification service, a general purpose facility for asynchronously delivering events from event producers to event consumers who have registered an interest in the events [7, 6, 5]. Siena utilizes a content-based addressing and routing scheme [7]. In content-based routing message destinations are not specified by the producer as they are

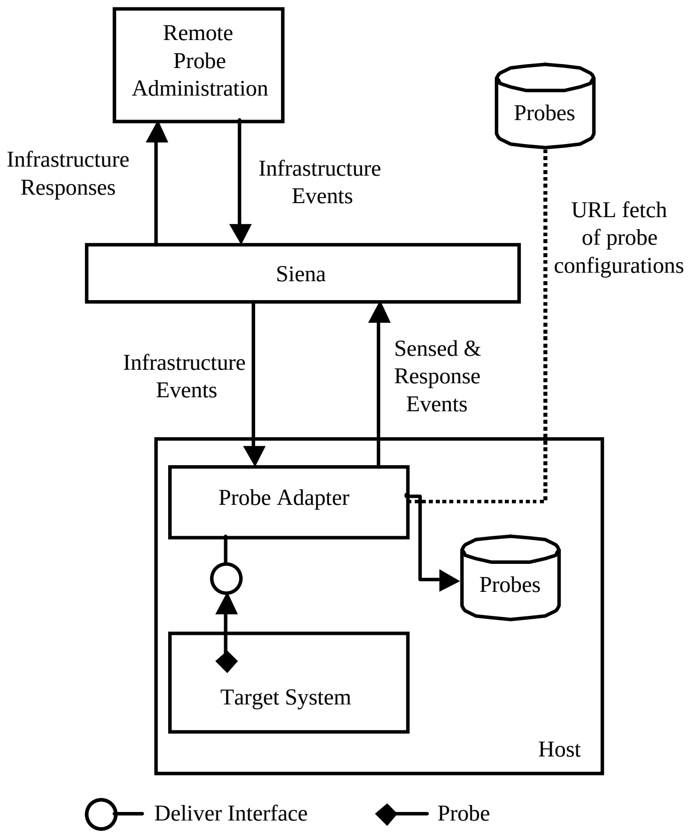


Figure 4.1: The Active Interfaces Probe Run-Time Infrastructure.

in typical unicast and multicast routing, instead consumers express interest in the events based on their content.

Siena event content is contained in notifications. Each notification is built up as a set of attribute-value pairs. Attributes are application defined and are identified by a string. Attribute values included in a notification can be strings, ints, longs, doubles, booleans or byte arrays [30]. Event producers publish notifications that contain sets of attribute-value pairs. Consumers use a subscription language to subscribe to notifications. The subscription language provides a set of operators (subset of SQL select query) that may be used on the attribute values of any attributes contained in a notification. The subscription language can be used to construct filters or patterns. Filters select single notifications by constraining the value of named attributes. Patterns specify a relationship between multiple notifications. If the pattern is matched all of the notifications which make up the match are delivered to the subscriber [6]. In addition to publish and subscribe Siena has an advertisement mechanism. An advertisement is essentially an intent to publish notifications that match a certain filter. In the most recent Java version of Siena 1.1.2, advertisements are not supported.

Siena events are distributed by a hierarchical network of event dispatching servers. The hierarchy is composed of instances of the `HierarchicalDispatcher` class. `HierarchicalDispatcher` is the primary implementation of the Siena event notification service [30]. `HierarchicalDispatcher`s can serve as Siena event service for local (same JVM) clients as well as remote clients. The hierarchy grows incrementally, when a new server comes on line it adds itself to an existing hierarchy by specifying a master that is already part of the hierarchy.

4.3.2 Infrastructure Events

The Active Interface Probe Run-Time Infrastructure supports all of the events presented in the high-level design. The events have been reworked to fit into the context of Siena Notifications. Each piece of information specified in the high level API is specified as a Siena attribute value pair. Table 4.1 provides an example Siena notification from our API.

In the high level design we specify that events should include as much information for the purposes of event routing as possible. We have include response as an event Type for explicitly that purpose. Entities in the monitoring system that wish to control probes on a system can subscribe to all response events emitted from that host and system. They could further refine their subscription to receive only response events from probe configurations in which they are interested. Had we not done this they would have had to either subscribe

Table 4.1: A deploy event and a deploy response event as they are represented as attribute value pairs in a Siena notification.

Attribute Name	**Attribute Value**
EventType	Deploy
Hostname	The name of the machine the target system is on.
SystemID	A string identifier of the software system to target.
ProbeConfigurationName	The name of the probe configuration that should receive this event.
ProbeConfigurationModule	The URL from which the configuration module can be downloaded. This should include either http: or file:
EventType	Response
ResponseType	DeployResponse
ProbeConfigurationName	The name of the probe configuration that sent this event
ProbeConfigurationModule	The URL that was included in the Deploy event
Hostname	The name of the machine from which the event is coming
SystemID	A string identifier of the software system to target.
ProbeConfigurationName	The name of the probe configuration that sent this event
ProbeConfigurationModule	The URL that was included in the Deploy event
Status	Success/Failure depending on whether the request was fulfilled
reasonForFailure	The inclusion of this attribute is conditional on the value of status being failure. A string giving some information about why the failure occurred. This attribute value pair is only included if Status is set to Failure

to all events that included the host and systemID as attributes or specifically subscribe to each of the response types (which would have been event types). In the first case listed above, the subscriber would receive other infrastructure events targeted at the host.

Response events also serve the purpose of reporting exceptional conditions. If the action specified in an infrastructure event fails, the Active Interface Probe Run-Time Infrastructure includes information in the response to help the entity that sent the infrastructure event understand why it failed. One possible reason for a failure is attempting to execute an action from a state where it is not supported (see Section 3.2). Another possible error is that the run-time infrastructure was unable to download the probe configuration module of a deploy command. There are a number of other possible errors that could arise in the user supplied probe configuration module.

4.3.3 Sensed Events

Sensed events are the events emitted by probes. We map the sensed events of the high level design into Siena notifications. Our sensed events must also adhere to the SmartEvents standard put forth by our collaborators at Columbia University [16]. SmartEvents define a FleXML schema to which events must conform in order to interoperate with portions of the monitoring system being developed by our collaborators. FleXML will be discussed briefly in the next chapter.

To satisfy these two different requirements we have designed a two-tiered sensed event. Since the probe adapter will publish sensed events as Siena Notifications, they will be routed based on their content. To support content based routing, the probe adapter will contain a number of attribute value pairs in the notification. These will include all of the information required by the high level design:

- host name
- system execution identifier
- probe configuration name
- probed class
- probed object identifier
- probed method
- probe location: before/after
- event type

In addition to the attributes included for the purposes of routing, the notification will include an event data attribute. The value for the event data will be the full FleXML SmartEvent. In addition to the attributes that were used for routing the SmartEvent contains a times-tamp and the name and string representation of each of the parameters to the method that triggered the probe.

```
<?xml version="1.0" encoding="UTF-8"?>
<smartevent xmlns=
          "http://www.psl.cs.columbia.edu/2001/01/fullAISchema.xsd">
     <metadata>
          <tag>99999</tag>
          <source>
               <ipAddr>130.215.28.24</ipAddr>
               <ipPort>1234</ipPort>
          </source>
          <time>.2000-11-20T19:02:00</time>
          <kxOpaque>true</kxOpaque>
     </metadata>
     <activeInterface>
          <!-- before or after function call?  -->
          <callbackType>BEFORE</callbackType>
          <!-- what is being instrumented?  -->
          <object>scheduler.meeting.MeetingBean@e56346</object>
          <class>scheduler.meeting.MeetingBean</class>
          <!-- method call trace -->
          <method name="deleteMeetings(MeetingDate,MeetingDate)">
               <param type="scheduler.meeting.MeetingDate">
                         8/31/00 9:30AM</param>
               <param type="scheduler.meeting.MeetingDate">
                         8/31/00 11:00AM</param>
          </method>
     </activeInterface>
</smartevent>
```

Figure 4.2: Example SmartEvent

To facilitate the generation of the SmartEvents we have created the ActiveEvent class. ActiveEvent has a constructor that requires all of the necessary information for the SmartEvent. Once constructed, a call to the objects `toXML` method returns a string representation of a

SmartEvent. ActiveEvents are created by the ProbeAdpater when it is building a sensed event in response to a method delivery from a probe.

4.3.4 Identifying Probes in the System

Active Interface monitors components at method boundaries. Callbacks can be inserted before or after method execution. It is thus natural to think of probes being identified in relationship to the component's methods. Likewise a component can be identified by the machine on which it is executing and a system id that can distinguish between different processes or JVMs. In the Active Interface Probe Run-Time Infrastructure this is how we handle probe identification. Any active interface probe can be uniquely identified by the combination of the following attributes: host name, target system, probe configuration name, probe type, class name, object identifier, method, and the string before or after.

4.3.5 Probes and Probe Configurations

As defined in the high level design, probe configurations are what the probe infrastructure handles. They are deployed activated etc. Each probe configuration is composed of one or more probes. Each probe monitors some portion of the target system. Probe configurations and probes developed by a third party must be able to operate in the probe run-time infrastructure. We have designed a set of interfaces that standardize the way probe configurations and probes will be handled in the AIPRI. In this section we describe the `ProbeConfiguration` and `HookControl` interfaces. `HookControl` is an interface that probes must implement. Additionally, we have created some classes which simplify the creation of new configurations.

The `ProbeConfiguration` interface (see Figure 4.3) contains exactly the methods necessary for a probe configuration to function fully in the AIPRI. Since a probe configuration is the final receiver of the events of the run-time protocol, many of the methods included in the interface mirror those of the run-time protocol. The others merit a little discussion here. First, each `ProbeConfiguration` must have a name and `get-Name` and `setName` methods. In the current implementation the name of the `Probe-Configuration` must be the fully qualified class name of the class that implements `ProbeConfiguration`. The `getClasses` method returns an `Enumeration` of `String`s. Each `String` is the fully qualified name of a Java class that is the target of a probe from this probe configuration. The `getProbes` method takes two arguments the first is the fully qualified name of the target class. The second is an object identifier. The probe configuration will return an `Enumeration` that contains the fully qualified class names of any of its `HookControls` that target either the named class or the named object. The object identifier is assumed to be the `String` returned by the target objects `getComponentName` method. This name is assumed to be more meaningful than the `String` returned by the `toString` method from `java.lang.Object`. In the future this name could be mapped from an architectural specification.

```
package dasada.probeAdapter;
interface ProbeConfiguration {
     public void activate( );
     public void deactivate( );
     public boolean isActive( );
     public boolean focus( String[] params );
     public String[] querySensed( );
     public Enumeration getClasses( );
     public String getName( );
     public void setName(String name );
     public Enumeration getProbes( String className,
                   String objectID );
 }
```

Figure 4.3: The `dasada.probeAdapter.ProbeConfiguration` Interface

It is clear that implementing the `ProbeConfiguration` interface from scratch would require quite a bit of work. To simplify this we have created an abstract base class, called `ConfigurationModule`. `ConfigurationModule` implements the `ProbeConfiguration` interface. Additionally it provides `addClassProbe` and `addObjectProbe` methods and supporting infrastructure to allow probes to be registered with the configuration. In order to create a fully functioning implementation of `ProbeConfiguration` it is necessary to extend `ConfigurationModule`. To provide any useful behavior the subclass must at a minimum override the init method and include in it method calls to the add probe methods in order to add probes to the configuration. The `focus` method in `ConfigurationModule` is abstract, so the user must override it. They can either include useful behavior or simply make the method a no-op.

`ConfigurationModule` includes a useful implementation of the `querySensed` method from the `ProbeConfiguration` interface. It builds an array of `String`s that identify each of the `ProbeConfiguration`'s active probes. Probe identification as we discussed in the last section is composed of: host machine, system identification, probe configuration name, probe name, class name, Object identification, method signature and before or after. All of this information is not available to the `ConfigurationModule`. So the building of the response is collaborative. The `ConfigurationModule` must determine what classes it probes and with what `HookControl`. Then each of the `HookControl`s are called to provide information about each method for which they are actively emitting events.

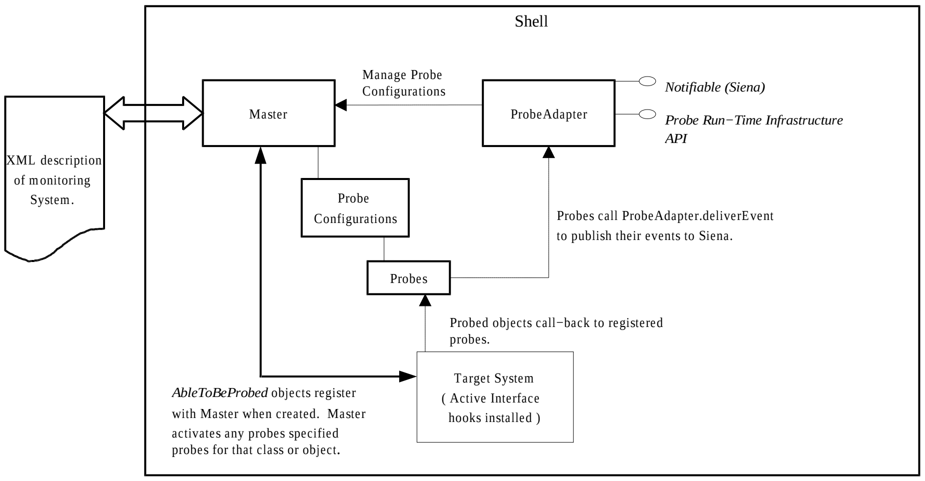
To allow the run-time infrastructure to handle probe insertion and removal in a standard way we have developed the `HookControl` interface. Figure 4.4 shows the interface. The `initCallbacks` method takes an `Adaptable` object and hooks up callbacks for it. The implementer of the `HookControl` interface must for determine what methods of the target class to probe and what methods to direct the callbacks to. This design decouples these decisions from the probe infrastructure and allows flexible monitoring. Likewise the `removeCallbacks` method is called to remove all of the callbacks connected in the `initCallbacks` method. We wished to defer the decision of whether to create a singleton probe or object probes to the implementer of `HookControl`. To allow both options we decided to use a static method to return instances of the `HookControl` object. If the probe developer wishes to have a singleton probe, they only create a single instance and return it to all callers of the instance method. Otherwise the `HookControl` will simply create and return a new instance each time the instance method is invoked. Java does not allow the inclusion of static methods in interfaces, therefore we have included the static `String instanceMethod`. This is the name of the no-argument instance method which can be used to get a `HookControl` instance. The set and get probe configuration name methods allow the `HookControl` to be informed what probe configurations they are associated with.

```
package dasada.probeAdapter;
public interface HookControl {
     public static final String instanceMethod =
                  "getHookControlInstance";
     public void initCallbacks ( Adaptable a );
     public void removeCallbacks ( Adaptable a );
     public void setProbeConfigurationName ( String s );
     public String getProbeConfigurationName ( );
     public String[ ] querySensed ( );
 }
```

Figure 4.4: The `dasada.probeAdapter.HookControl` Interface

4.3.6 Design

In the high level design the probe infrastructure was described as consisting of a probe adapter and probe configurations. The probe adapter was responsible for facilitating the communication of the monitoring system with the probes and providing general probe control mechanisms. We have distributed the tasks assigned to the probe adapter to several classes and developed other infrastructure necessary to support the run-time deployment and management of probe configurations. Figure 4.5 presents the Active Interface Probe Run-Time Infrastructure (AIPRI).



Shell
Manage Probe Configurations
Master
ProbeAdapter
Notifiable (Siena)
Probe Run−Time Infrastructure API
XML description of monitoring System.
Probe Configurations
Probes call ProbeAdapter.deliverEvent to publish their events to Siena.
Probes
Probed objects call−back to registered probes.
Target System (Active Interface hooks installed)
AbleToBeProbed objects register with Master when created. Master activates any probes specified probes for that class or object.

Figure 4.5: The Active Interface Probe Run-Time Infrastructure. The monitoring system and the target system run within a shell. The `ProbeAdapter` implements the Siena interface so that it can publish events and subscribe to infrastructure events. `Master` serves as a repository for all deployed probe configurations and provides Java APIs for interaction with them. `Master` maintains an xml document describing the status of the probe configurations within deployed to the system. The XML document survives machine shutdown and allows the Probe Infrastructure to restart in the same state when the system is restarted.

The `ProbeAdapter` class functions as a mediator [15] between the probe infrastructure and the monitoring infrastructure. The `ProbeAdapter` implements the Siena `Notifiable` interface so that it can subscribe to infrastructure events targeted at the probe configurations it manages. The `ProbeAdapter` has as a member a Siena `HierarchicalDispatcher`. This `HierarchicalDispatcher` acts as the local implementation of the Siena event notification service. Events from the monitoring system are delivered through it to the `ProbeAdapter` and the `ProbeAdapter` can deliver events through it to the monitoring system. Within the bounds of the `Notifiable` interface the `ProbeAdapter` implements the protocol described above.

`Master` essentially implements the probe run-time infrastructure API in Java as opposed to Siena Notifications. It provides methods that allow for each of the operations of the API. `Master`'s responsibilities are further divided among a set of helper classes. `Master` is responsible for maintaining a persistent view of the probe infrastructure that it oversees. It does this by maintaining information that defines what probe configurations are deployed, installed, or activated. When the system is started it reads in the information and sets up the system as it is described.

`Master` is also the self registration authority for the AIPRI. Each `Adaptable` and `AbleToBeProbed` object that is created in the target system will call masters static register method. If there exists a probe configuration in the active state that is designed to monitor objects of that class or specifically that named object, the probe configuration's probes will be hooked up to the object. If no probe configurations exist that target the particular class or object, `Master` keeps a reference to the `Adaptable` object in a registry. If at a later time a probe configuration that monitors the object is deployed and activated master will hook up probes to it. It is important to note: the object reference is stored in a `java.lang.ref.WeakReference` object. No strong references to the object are maintained by the probe infrastructure. This allows correct garbage collection of the object if its reference goes out of scope in the target system.

`Master` is assisted by a number of helper classes not described here.

4.3.7 Loading Probe Configurations

Probe configurations must be able to be deployed at run-time. As specified in the API, the deploy event contains a URL from which the configuration can be downloaded. `Master` will download the named file and save it locally. Since a probe configuration will be made up of an implementer of the `ProbeConfiguration` interface and at least one `HookControl`, we have decided to require that all of the classes required for use in the probe configuration be packaged in a JAR file.

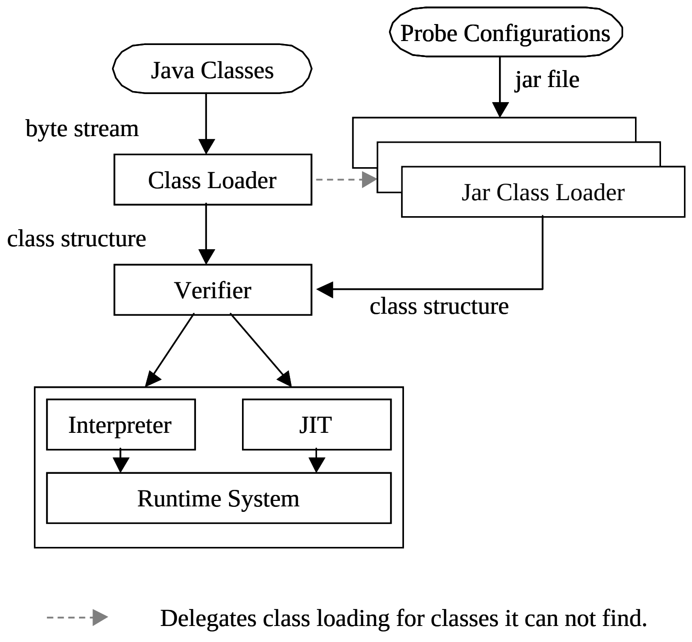


Figure 4.6: The AIPRI uses multiple instances of a jar loader we developed. Each probe configuration is explicitly loaded from a new instance of the jar loader. Subsequent classes created from inside the probe configuration are loaded through the Java 1.2 delegation mechanism.

We have implemented a jar loader that can be used to directly load the probe configuration and probes from the jar file that was downloaded. The jar loader implements the Java 1.2 class loading delegation scheme. We envision the monitoring system to have multiple probe configurations deployed at any given time. We also feel that it is a reasonable assumption that probe configurations will be developed by multiple independent groups. We have therefore decided that each probe configuration will be loaded by a different jar loader. Each probe configuration is explicitly loaded by a new jar loader with the command `Class.forName` method from `java.lang.Class` which allows the specification of a class loader. Classes loaded from inside the probe configuration can be constructed in the normal manner but their loading is delegated to the class loader which loaded the class they are being created in. Figure 4.6 shows the delegation class loading scheme. Using separate jar loaders allows us to define separate name domains within which the probe configurations can function without fear of unexpected interactions with code they do not know exists. Multiple probe configurations can utilize the same classes without fear of name conflicts. The only possible name collisions that can occur are with the name of the probe configurations themselves. The probe infrastructure has been written so that probe configuration names must be unique.

4.3.8 Bootstrapping

It is necessary for the Run-Time infrastructure to be present in the same Java Virtual Machine as the target system, and furthermore for the run-time infrastructure to start first. This need has been met by the creation of a generic bootstrap wrapper class: `dasada.Shell`. `Shell` is passed the name of and arguments that are required to start a Java program. It then starts the run-time infrastructure and calls the `main` method of the target application with the provided parameters.

4.4 Active Interface `ComponentAdapter` Configurations

There are a number of possible configurations that can be used for the setup of active interface probing (see Figure 4.7). Essentially there are two variables in the setup. The first is whether to allow/utilize singleton or individual instances of `ComponentAdapter`. The second is whether to use static or object probes. These choices affect the simplicity and flexibility of the monitoring that is possible.

The AIDE compiler inserts the code of the `Adaptable` interface into the target component. This includes the private `ComponentAdapter` variable and the public accessor methods that allow it to be gotten and set. Since the `ComponentAdapter` variable is not static, to accomplish a single `ComponentAdapter` for all objects of a target class it would be necessary to use a single instance of `ComponentAdapter` for all object instances. The self registration authority (`Master` in the AIPRI) could be written to simply assign the same adapter to all instances of a particular class.

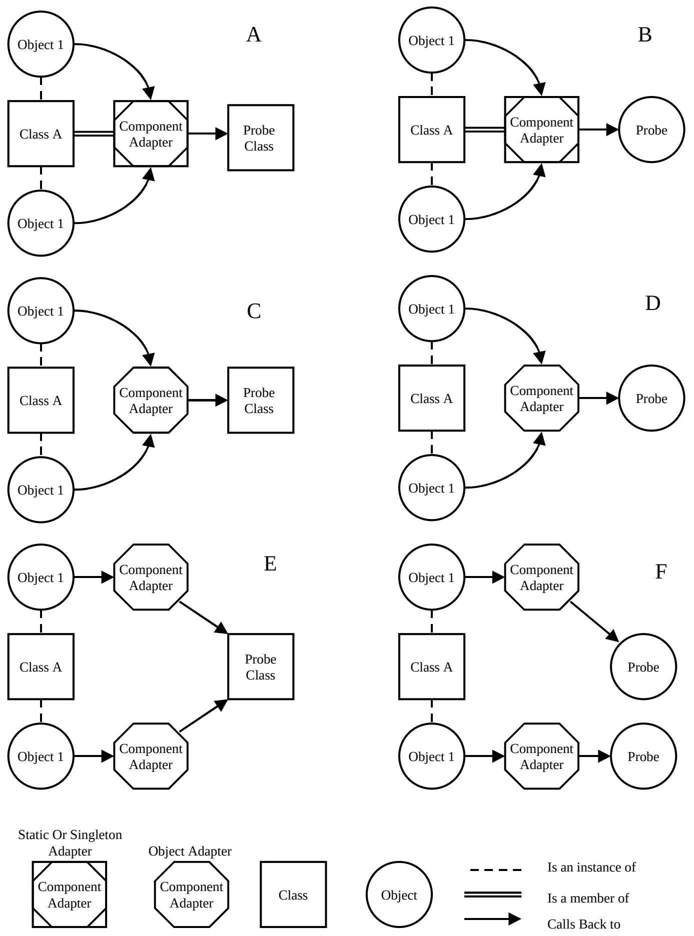


Figure 4.7: Active Interface `ComponentAdapter` Configurations. In all of the cases shown it is possible for more than one probe to be associated with a particular `ComponentAdapter`.

The possible cases range from the simplest case in which there is a single `ComponentAdapter` for all objects of a target class and a single static probe (see Figure 4.7.A) to the case where each object has its own `ComponentAdapter` object and its own instance of a probe (see Figure 4.7.F). In the first configuration all classes would have exactly the same methods probed. The probe would be unable to simply maintain state related to the objects that they probe. Any configuration using static/singleton probes is probably not appropriate for use by hybrid active interface probes. In the later case different objects could have different methods probed and the individual probes can maintain state related to their target object. This configuration is the most flexible an most useful for hybrid probes.

The Active Interface Run-Time Infrastructure has been designed to work with object adapters and object probes. It does not support any of the possible configurations that utilize a singleton adapter. Also, instead of supporting static probes it supports singleton probes. It defers the choice between singleton probes and instance probes to the designer of the probe configuration. This is accomplished by the requirement that `HookControl` implementers must provide a static `getHookControlInstance` method. If the probe configuration designer wishes to maintain any target object specific state inside the probe, object probes would be the obvious choice. If however, the probes are intended only as event emitters a singleton probe would be a reasonable choice and would simplify the implementation of the `focus` and query sensed methods.

4.5 Specific Active Interface Probes

In this section we first describe a `HookControl` that can be used to automatically hook up callbacks for all of the methods of a target class. We then present other possible uses of Active Interface probes. As we will see in the next chapter some of these applications overlap with those of gauges. Designers of the monitoring system can choose where to implement these activities. By implementing them in the probes, event traffic to the monitoring system is decreased. However, if there are multiple consumers for a particular event type it may be desirable for the events to simply be emitted to the monitoring architecture and for any processing or filtering to be done by subscription or by gauges.

4.5.1 Automatic Probes

To simplify the process of creating `HookControl` instances we have created an automatic probe called `AllProbe`. Each `HookControl` is responsible for connecting the AIDE inserted hooks in the component with probe methods (these would typically be in the `HookControl` object). Callbacks are hooked up through the target component's adapter using the method `insertHook`. The `insertHook` method requires as parameters `String`s describing the method signature of the method to be probed, the name and parameter types of the method that will be called and an object reference to the probe object. Traditionally, these method calls have been constructed by the adaptation implementer. In `AllProbe` we use Java's reflection capabilities to automatically insert before and after callbacks for each of the methods of the target object.

There are two probe methods in `AllProbe`. The first, `methodCalled` is attached to as the before callback for each probed method. The second, `methodReturning` is attached for each after callback. `AllProbe` is designed as a generic event emitter. It performs no processing. It simply delivers an event encapsulating the method call that lead to the probe trigger.

There are a number of inheritance related issues that `AllProbe` must take into account. In order to hook up meaningful callbacks there must be Active Interface hooks present in the source code for the methods probed. A Java object will include methods that it inherited from its super classes. It does not make sense to attach probes for the methods of the super classes unless the individual performing the probing knows that the super class has been Active Interface enabled. Since `AllProbe` is intended to be a generic probe, it does not attach callbacks for any of the methods of the super classes. `AllProbe` keeps track of what method calls it has inserted. A call to `removeCallbacks` results in the unhooking of all of the callbacks that were inserted.

In addition to simplifying the insertion and removal of callbacks, `AllProbe` defines a useful implementation of the `focus` method. `AllProbe`'s `focus` method allows a user to shutdown the events being emitted from certain method calls. As described previously in this chapter the `focus` method takes an array of `String`s. To stop (or start) events for a certain method the user would pass in an array in which the first element is the `String` “deactivate” (“activate”). The second element would be the signature of the method to deactivate. The `focus` method supports the activation or deactivation of multiple methods at once. Each method to be changed should simply be included as an additional element in the `String` array.

`AllProbe` also provides an implementation for `querySensed`. It will return information only for those methods that are currently activated.

`AllProbe` provides a very fast way to create a probe configuration. By extending `ConfigurationModule` and using `AllProbe` a developer can begin to generate monitoring information by writing approximately ten lines of simple code. The use of `AllProbe` in the probing of GeoWorlds will be discussed in the next chapter.

4.5.2 Constraint Checking Probes

Constraints are conditions that must be met for a method to function correctly. Constraints are similar to the preconditions, postconditions and invariants of the Eiffel programming language [14]. In Eiffel a component designer would formally specify preconditions, postconditions and invariants. If the resulting application is run with monitoring switched on violations of the conditions will cause assertions to throw an exception. The component developer can define exception handlers that fix the issue and allow continued operation or they can allow the exception to cause the program to halt [14].

Active Interfaces provide a mechanism through which constraints could be implemented as hybrid probes. The implementer of a before callback in active interfaces can check the values of the parameters to the method. If a precondition constraint is violated the callback code can return deny and the method will return without executing its code. In addition to this an after callback can be used to check post conditions and invariants. Possibly restoring the pre-invocation state if the constraints are violated.

For the purposes of monitoring we generally do not wish to intercede in the target systems operation. Instead of denying the method invocation the callback code could publish an event indicating that a constraint has been violated. In a system utilizing the Active Interface Probe Run-Time Infrastructure it would be simple to develop a probe for which constraints could be changed dynamically at run-time.

4.5.3 Hybrid Active Interface Probes

As defined in the probing chapter, hybrid probes combine some aspects of passive triggered probes and actively triggered probes. An Active Interface hybrid probe could collect information in response to the normal before an after callbacks from the target system. In response to information requests, or on a schedule, a summary of the information could be packaged as an event and sent out to the monitoring infrastructure.

A particularly useful application of this technology would be the creation of probes that monitor a high frequency event in the target system. Instead of bombarding the monitoring system with each of these events a count or summarization would take place and be delivered to the monitoring system on a schedule. Alternatively, the individual events could be stored for some period of time and provided to the monitoring system in response to specific requests.

4.6 Demonstration of the Active Interface Run-Time Infrastructure

We have created a demonstration of the Active Interface Run-Time Infrastructure. The purpose of the demo was to test the Active Interface Probe Run-Time Infrastructures ability to deploy and interact with probes in a system at run-time.

The previously existing target system for the demonstration is a Client-Server Dictionary. Clients send the server a `String`. If the `String` is a word the Dictionary server returns a value indicating that the word is valid. If the `String` does not represent a word the dictionary returns a `String` indicating that it was not a word. The system is written in Java and uses TCP/IP for communication.

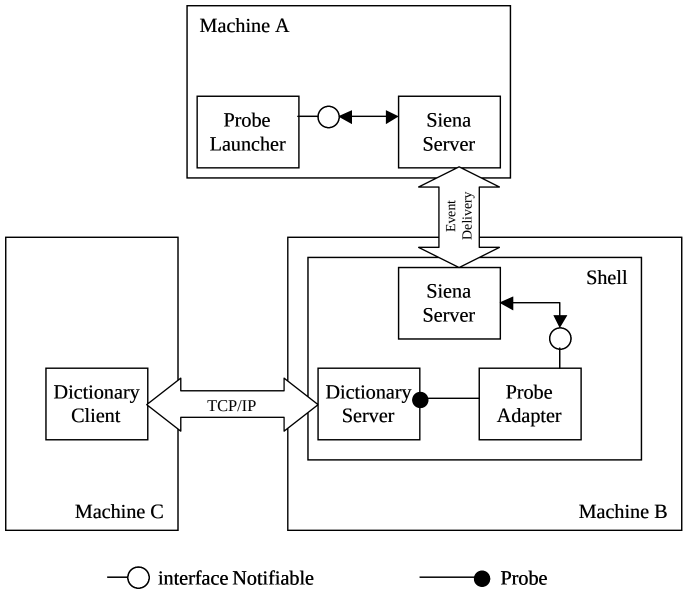


Figure 4.8: The Dictionary Demo of the Active Interface Probe Run-Time Infrastructure.

We developed a probe configuration called `DictionaryConfigModule` by extending the `ProbeConfigModule` class. In the init method of the `DictionaryConfigModule` we add a single `HookControl` called `Timer` as a class probe of all objects of the class `Dictionary`. It also implements the abstract method `focus` from the `ProbeConfigModule` class. The `focus` implementation is designed to pass a parameter to a static `focus` method in the `Timer` `HookControl`.

The key methods of `HookControl` are `initCallbacks` and `removeCall-backs`. When `Timer`'s `initCallbacks` method is called with the `Adaptable` object `Dictionary` as its argument, two callbacks will be hooked up. One before the `Dictionary`'s `isValid` method and one after it. When later invoked the before callback gets the current time and delivers an event encapsulating information about the method invocation to the `ProbeAdapter`. The after callback again gets the current time. It then calculates the time since the before callback and includes this information in the event it delivers to the `ProbeAdapter`.

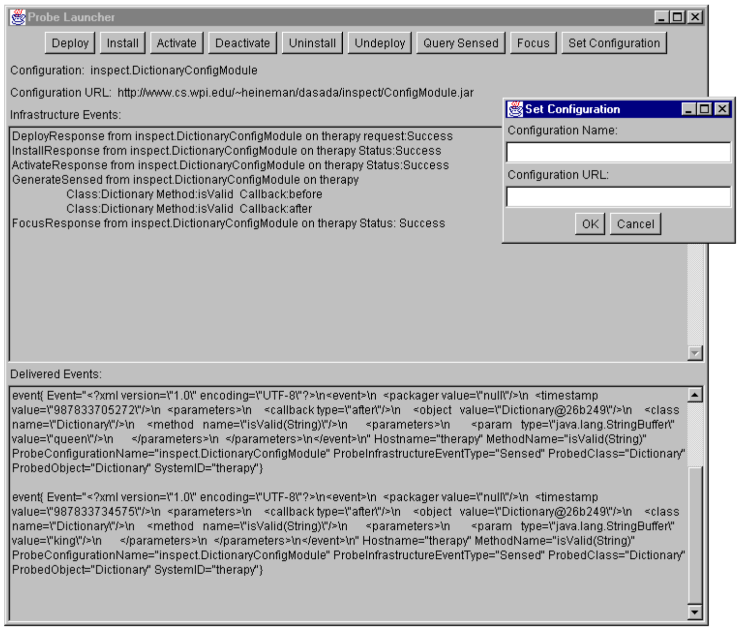
`Timer` has two methods in addition to those of the `HookControl` interface. The first `methodInvoked` is used for before callbacks. It gets the current time when it is invoked and delivers an event to the `ProbeAdapter` for publication into Siena. The second method, `methodReturning` is used for after callbacks. It again gets the time and calculates the time spent fulfilling the request. It then delivers an event encapsulating this information to the `ProbeAdapter`. `Timer`'s static `focus` method can accept either of two `String`s to have a meaningful result. If the `String` "summary" is passed in the `Timer` quits delivering before events to the `ProbeAdapter` and only delivers the after event. If the `String` "beforeAndAfter" is passed in `Timer` will deliver both before and after events. Any other `String`s will result in a response event that reports the status as failed.

To give an idea of the simplicity of implementing the `ProbeConfiguration` interface by extending `ProbeConfigModule`, `DictionaryConfigModule` contains sixteen lines of code (including method signatures and such). Only three of these lines of code are actually specific to the `DictionaryConfigModule`. `Timer` is a little more involved. It required around seventy lines of code. Approximately two thirds of this is cookie cutter code that would be significantly similar for any `HookControl`, the remainder was the actual probe insertion code and the `focus` method.

This setup was intended to be trivial. It does not handle the possibility of multiple overlapping requests to the dictionary. However, Active Interfaces provide all of the necessary information to distinguish between method invocations, so it could easily have been updated to do this.

A `ProbeLauncher` and graphical user interface were created to launch and interact with the probes at run-time (see Figure 4.9). The `ProbeLauncher` implements the interface `Siena.Notifiable`. It registers to receive all probe infrastructure events and all sensed events published by the target system. The set configuration button allows the user to set an active `ProbeConfiguration`. Clicking on any of the remaining buttons results in the publication of an infrastructure event targeted at the active probe configuration. The top text area in the GUI displays response events that are generated for the infrastructure events published. The bottom window displays sensed events that were delivered from the deployed probes. The GUI simplified the testing of the AIPRI by allowing the dynamic deployment and control of probe configurations. The set configuration dialog box (open in 4.9) allows the user to enter the name and URL for a new probe configuration. All subsequent events target the configuration the user enters. The launcher was initially created for this demonstration but we continued to use it for testing purposes in the development of the demonstrations of the full monitoring system.

The dictionary server is started in the Shell which starts the AIPRI. The dictionary server runs as normal fulfilling requests from any clients. When the deploy event is received by the AIPRI the probe configuration named in the event is downloaded to the probe adapter and stored on the local host's disk. When the probe configuration is activated it monitors the time required by the server to search for the words sent by the



```
Configuration:  inspect.DictionaryConfigModule

Configuration URL:  http://www.cs.wpi.edu/~heineman/dasada/inspect/ConfigModule.jar

Infrastructure Events:
DeployResponse from inspect.DictionaryConfigModule on therapy request:Success
InstallResponse from inspect.DictionaryConfigModule on therapy Status:Success
ActivateResponse from inspect.DictionaryConfigModule on therapy Status:Success
GenerateSensed from inspect.DictionaryConfigModule on therapy
            Class:Dictionary Method:isValid  Callback:before
            Class:Dictionary Method:isValid  Callback:after
FocusResponse from inspect.DictionaryConfigModule on therapy Status: Success

Delivered Events:
event{ Event="<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<event>\n  <packager value=\"null\"/>\n  <timestamp
value=\"987833705272\"/>\n  <parameters>\n    <callback type=\"after\"/>\n    <object  value=\"Dictionary@26b249\"/>\n    <class
name=\"Dictionary\"/>\n    <method  name=\"isValid(String)\"/>\n      <parameters>\n      <param  type=\"java.lang.StringBuffer\"
value=\"queen\"/>\n      </parameters>\n  </parameters>\n</event>\n" Hostname="therapy" MethodName="isValid(String)"
ProbeConfigurationName="inspect.DictionaryConfigModule" ProbeInfrastructureEventType="Sensed" ProbedClass="Dictionary"
ProbedObject="Dictionary" SystemID="therapy"}

event{ Event="<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<event>\n  <packager value=\"null\"/>\n  <timestamp
value=\"987833734575\"/>\n  <parameters>\n    <callback type=\"after\"/>\n    <object  value=\"Dictionary@26b249\"/>\n    <class
name=\"Dictionary\"/>\n    <method  name=\"isValid(String)\"/>\n      <parameters>\n      <param  type=\"java.lang.StringBuffer\"
value=\"king\"/>\n      </parameters>\n  </parameters>\n</event>\n" Hostname="therapy" MethodName="isValid(String)"
ProbeConfigurationName="inspect.DictionaryConfigModule" ProbeInfrastructureEventType="Sensed" ProbedClass="Dictionary"
ProbedObject="Dictionary" SystemID="therapy"}
```

Figure 4.9: The Probe Launcher GUI.

client.

The demonstration successfully showcased the ability of the AIPRI to handle the dynamic deployment and control of active interface probes to a running system.

4.7 Conclusions

In this chapter we have presented the design of the Active Interface Probe Run-Time Infrastructure. We showed how it aligns with the high level design and how it can be used to dynamically add new probes to a running system. We presented the design of an automatic probe that greatly simplifies the work of a system instrumenter. We then described a demonstration of the Active Interface Run-Time Infrastructure that was used to introduce our collaborators to its use. In the next chapter we will discuss the integration of the system presented here with the full monitoring architecture being developed collaboratively with Columbia University's Programming Systems Laboratory.

Chapter 5

Monitoring Architecture and GeoWorlds Demo

5.1 Introduction

In this chapter we describe a complete monitoring system that is being developed in collaboration with Columbia University. We briefly describe the components of the system, how they interact and how the Active Interface Probe Run-Time Infrastructure was integrated into the system. We introduce the notion of a gauge and present several gauges. We then describe the target system that will be used for a demonstration of the complete monitoring architecture. We describe the development of probes to collect information from this system and finally we describe gauges that we have developed for use in the demonstration.

5.2 Kinesthetics eXtreme: KX

Kinesthetics eXtreme (KX) is a dynamic system for run-time monitoring of the functional and extra-functional properties of component-based systems (see Figure 5.1). It is composed of three main parts: probe infrastructure, event infrastructure, and gauge infrastructure.

The probe infrastructure includes the various types of probes, the mechanisms for inserting probes into a target system, and policies for how the probes can be controlled by the monitoring system. The Active Interface Probe Run-Time Infrastructure described in the last chapter has been selected as one of two primary implementations of the probe infrastructure for this system. The other is worklets a technology being developed at Columbia.

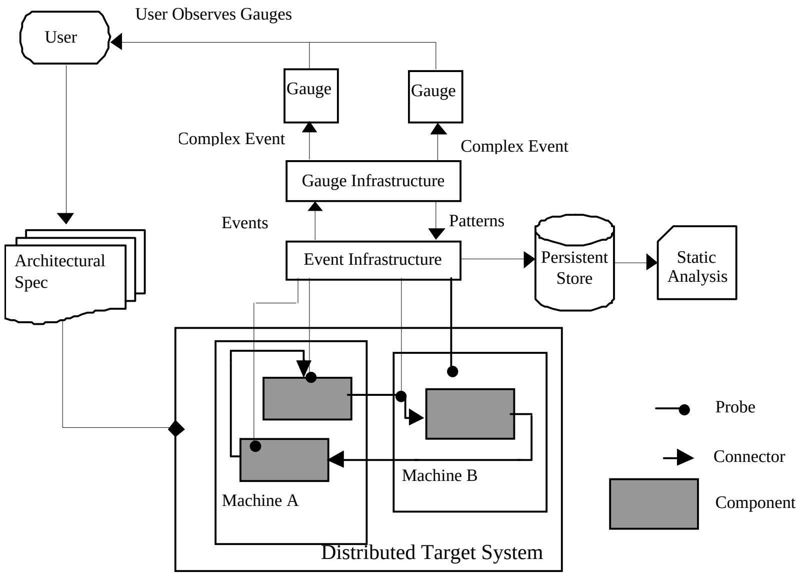


Figure 5.1: Caption goes here.

In KX, when a probe is triggered it will generate a FleXML event. FleXML is an XML-like formalism with extensions that allow for partial ordering, incremental handling of documents as they become available and discovery of how to handle previously unknown tags [20].

The event infrastructure was primarily developed by Columbia University. It is composed of an event standard, an internet scale event bus, a pattern language, a pattern matching facility and an event storage facility. The event standard being used is the Smart Events Schema [31]. Events will contain type, time stamp and location. Specific event types may be defined that include additional information. The XML-based Universal Event Service (XUES) overlays Siena [20]. Siena provides a distributed event bus supporting publish/subscribe and simple filtering services through which events are disseminated to interested parties [5, 6]. XUES combines three main entities: an event packager, an event distiller and an event notifier. On entry into XUES events are processed by the event packager. The event packager is responsible for encapsulating events in smart events and Siena notifications. The packager also saves the events into a database. The base attributes stored for each event are its source, time of receipt, type of data, and the actual data. Events do not persist in the database indefinitely, instead there will be an expiration mechanism. The event distiller supports pruning of the dataset and the detection of meta-events(patterns of single events). The event distiller instead of being a single component of the design is a subsystem. Its activities are built on top of a second private Siena event bus. Events from the system wide event bus are transferred to the internal bus [12]. The internal event bus has been used to allow the distribution of the pattern matching to a number of sites and to take advantage of its subscription mechanisms to provide a first pass filter. The state machines are constructed from rules specified in XML documents. When a state machine detects a match to its pattern a meta-event is created. The individual events comprising the meta-event are encapsulated in a single event and published to the external Siena bus [12]. Notifiers register to receive these meta-events. Furthermore, the meta-events are published back to the internal bus. This allows other state machines to be specified that consume meta-events. The XML documents that are used to create the state machines are created and passed into the event distiller by event notifiers. Event notifiers deliver notifications that the meta-event has been received onto gauges, semi-persistent storage and other interested parties.

In KX, gauges subscribe for notifications of single events or patterns of events. The gauge infrastructure provides a framework for the creation and execution of gauges. At this time it consists primarily of the language used to create patterns. In a future version of KX gauges will be integrated into TRiKX (see next paragraph). Gauges will in general be developed by domain experts and inserted into the gauge infrastructure. Each gauge will then subscribe to the events in which it is interested. If a gauge is interested in a complex event it will upload the pattern of events that comprise the complex event to the event infrastructure. In the initial prototype, gauges will be observed by human operators who will make decisions based on their output. The gauges we describe as part of our demonstration do not integrate completely with the KX gauge infrastructure as it was not available in time to allow integration prior to the June demonstration. Instead our gauges subscribe directly to the primary Siena bus for delivery of raw events from the probes. Any pattern matching or processing required is performed by the gauges. The KX architecture is built so that in later work gaugents (gauge + agent) will make decisions and take action based on the event patterns they monitor.

5.3 Integrating the AIPRI with KX

By using the model/view/controller design pattern in the design of KX, integration issues have been greatly simplified. Probes are the part of the controller that interacts with the model (the target system). The remainder of the controller is composed of the Siena event service and the Smart Events schema. The controller in the model view controller allows decoupling of the model and the views. Siena's publish/subscribe mechanism accomplishes this. The AIPRI probe adapter provides an interface to the Siena event services. Integration simply required connecting the probe adapter to the appropriate Siena server hierarchies. The AIPRI creates and emits smart events. Events sensed by probes in the AIPRI can therefore be delivered through the Siena bus directly into KX.

Events emitted by the probes will automatically have two destinations in the monitoring system. The first is the event packager. The event packagers primary function is to translate non Siena events for consumption by the rest of KX. AIPRI events are already delivered in Siena format so this step is not necessary. The primary function is not needed by events emitted from the AIPRI since the events are emitted directly as smart event bearing Siena notifications. In addition to this function the packager provides persistent storage facilities. Persistent storage is useful for post-mortem understanding of the events that led to a failure. It also may be useful to enable refinement of the understanding of how service quality has degraded over time.

5.4 Gauges

Gauges are the consumers of probe output. They are used to process the incoming events into meaningful information and relate the information to decision makers in a useful fashion. Decision maker here need not refer to a person, it could instead indicate a software entity that will make decisions based on the provided information. In the remainder of this section we present several gauges and describe the nature of the probes that would support them. In the GeoWorlds Gauges section we discuss actual implementations of several that form part of the GeoWorlds Demonstration.

5.4.1 Performance Gauges

Performance gauges can be applied to a number of issues. The most obvious of these is the determination of how long a process takes and where most of the time is spent. This determination can allow developers to focus their energies where they will be most beneficial. This has largely been the realm of specialized systems for the performance analysis and tuning of parallel systems or has been achieved through the use of post-mortem tools like prof and gprof [32]. These tools have often been operating system or hardware dependent. With active interfaces and the KX architecture we can bring performance monitoring to general software systems at run-time.

A general performance monitoring gauge would register for all events coming out of a particular subsystem. Each event will have a time stamp included by the probe run-time infrastructure. As the gauge receives these events it calculates the time difference between the related events that occurred before and after a method call. This information can be aggregated on a per method basis and on a per class/object basis. A gauge could have multiple views showing where time is spent. A default view shows how much time is spent in each class or object. The user can click on the gauge to show a detailed break down of which particular methods are using the most time.

We suggest specifically targeting distributed system performance. Many distributed programming frameworks attempt to make location transparent to the programmer. Remote objects or services are accessed as if they were resident on the local machine. This approach makes programming distributed systems considerably easier. However, by abstracting away location they also remove the programmers consciousness of the possible consequences of remoteness. Performance gauges can be used to highlight the impact of interacting with remote components.

Distributed programming frameworks often make use of stubs to allow the local program interact with remote services in a straight forward way. By specifically probing these stubs we can collect information about distributed service performance. In this application it would be useful to additionally identify the host of the remote system. It might be possible that a particular host was introducing long delays. If a different host existed on which the service could run, the gauges could oversee to the reconfiguration of the system.

5.4.2 Constraint Violation Gauges

In the Active Interface Probe Run-Time Infrastructure chapter we presented constraint monitoring probes that could monitor the preconditions, postconditions and invariants of a method. In that section we presented the constraint checking as a possible use of hybrid probes. If hybrid probes are used to perform the constraint checking and only emit events when constraints are violated, a constraint monitoring gauge would simply need to display what method invocations have violated constraints. However, constraint checking could be performed by gauges on events emitted from general probes as well. In that case, the gauge would need to register for all events emitted by certain method calls. The gauge itself would then compare parameter values against constraints. The two possibilities have the exact same outcome. They offer the gauge designer a trade off. When probes implement the constraints fewer events enter the monitoring system. If there are other parties interested in those events it makes sense for the constraint checking to be performed by gauges.

5.4.3 Failure Isolation Gauges

Failure isolation gauges work in two stages. First they must localize the source of a failure in the system. After the failed module is localized it can be isolated instructing the probe at the isolation point to use the active interface deny mechanism. In order to do this the developer must have a deep understanding of the system. Simply isolating any failing component could have disastrous effects on the software system. However the first stage, failure localization can be utilized with out risk and possibly with significant benefit. We therefor present failure localization gauges as independent entities. Following this presentation we discuss the full isolation gauge. Finally we discuss the circumstances that must hold true for these gauges to work at run-time.

Failure localization requires the insertion and activation of probes at key locations in the code (as determined by someone familiar with the target system). Each probe is set up to emit before and after events. Faults are localized by pairing before and after events from the probes. If a failure occurs, or an untrapped exception is thrown the events will not form pairs. The incomplete event pair deepest in the call tree represents the localization of the failure. By examining the parameters in the last successful event it may be possible to determine the reasons for the failure. It is important to note that a trapped exception will result in some unmatched pairs. However, at the location in the code where the exception is caught a matching return event will be generated. To facilitate failure localization we have designed a probe that includes additional information along with that included to describe the method call that triggered the probe. Specifically it includes the thread id of the thread that is executing in the probe code and a serial number which will be incremented after use. With active interfaces the thread in the probe is the same as that which was executing in the probed method. These two additional parameters can be used to help correlate events.

Once the failure has been localized it may be possible to isolate that portion of the call tree. Active Interface allows the callback method to deny the invocation of the target method by returning an exception object. The probe could be instructed to deny or override any further invocations of the errant method. If the reason for the failure was traced to a certain parameter value or range of values the probe could only deny the method only if the errant parameter recurs. Active Interface also allows the overriding of a method by returning an override object. Active Interfaces provides another possibility. If the probe developer was able to locate an alternate implementation of the desired functionality they could override the erring method. In this situation the probe would invoke the new implementation and return an override event that encapsulated the correct result. The Active Interface hook in the target method would return that result and prevent the execution of the original method.

Failure localization can be run post-mortem in any failure situation. Run-Time failure localization and failure isolation can only work in certain circumstances. Specifically the system must be able to recover from the initial error failure. Active interface provides no recovery mechanism. While this is a limitation it does not totally destroy the potential of this type of monitoring. The most likely situation in which this type of monitoring would be useful is in an asynchronous invocation situation. In that type of system the decoupling provides greater possibility for initial recovery.

5.4.4 Experience Based Expectation Gauges

Experience Based Expectation gauges monitor some aspect of a computation over time and develop a history of its observed behavior. Each time the computation occurs new results are added to the history. If the computation deviates from its average behavior it is flagged and brought to the attention of decision agents. EBEs provide a very useful subset of performance gauges but their use is not restricted to performance in the traditional sense. Any computation that produces a result that can be quantified statistically with respect to its correctness or desirability is a suitable target for EBE gauging. We suggest that an EBE could be used to monitor inputs to a method. Inputs that fall far outside the range of typical values could be indicative of a problem in the calling module. Other potential targets of EBEs include: cache hit rate, the number of results returned by one search engine relative to the number returned by others, database table query rates.

5.5 The Target Application: GeoWorlds

GeoWorlds is a component-based information management system. It integrates distributed information sources with geographic information systems and information processing services. It is intended to allow a user to quickly assemble, analyze, display and share information about a region, its resources and some particular activity. The system has been demonstrated for use in disaster management, intelligence analysis and scientific collaboration. GeoWorlds is comprised of custom components, CoTS components and publicly available web-based services [9]. GeoWorlds has been designed to simplify the addition of new services.

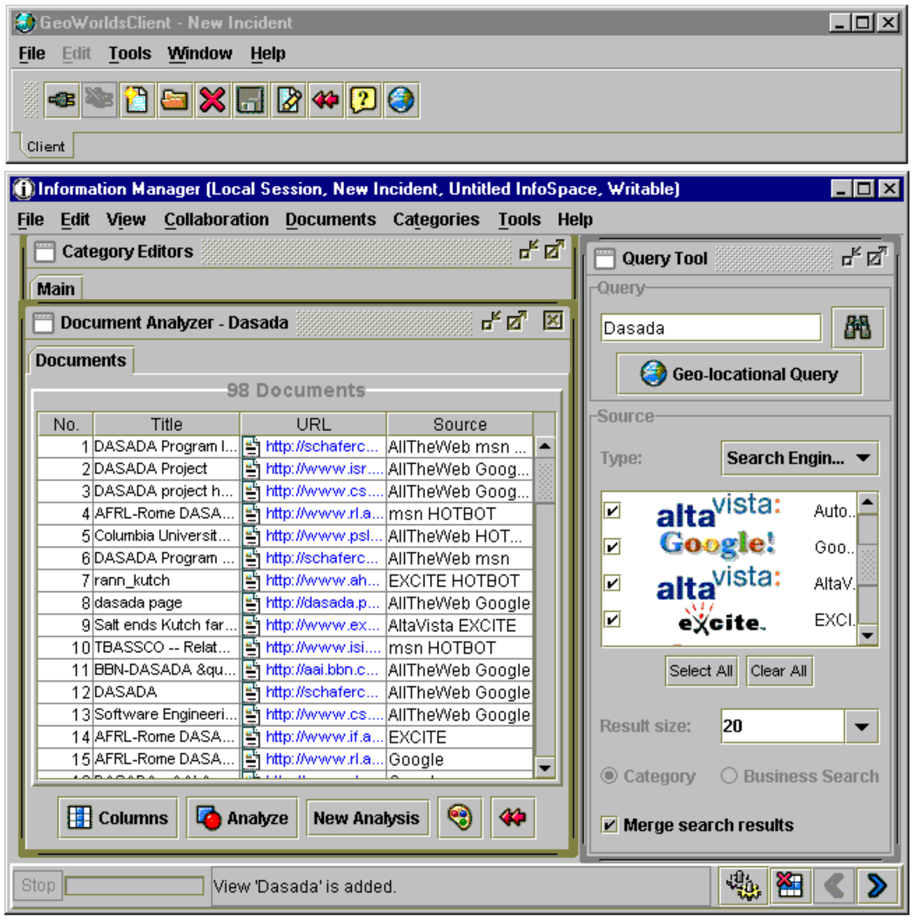
The distributed nature of the services utilized and the integration of commercial components makes GeoWorlds a good candidate for a demonstration testbed for the continual validation system.

Of particular interest for monitoring is the service launching architecture of GeoWorlds 5.3. The Information Manager is called Dasher.

5.6 Preparing GeoWorlds for Monitoring

There are five steps necessary to prepare GeoWorlds for monitoring in by KX.

1. GeoWorlds must be launched by the AIPRI shell.
2. Appropriate target classes must be identified.
3. The target classes must be prepared for probing by AIDE.
4. Probes must be developed.


GeoWorldsClient - New Incident
File Edit Tools Window Help
Client
Information Manager (Local Session, New Incident, Untitled InfoSpace, Writable)
File Edit View Collaboration Documents Categories Tools Help
Category Editors
Main
Document Analyzer - Dasada
Documents
Query Tool
Query
Geo-locational Query
Source
Select All
Clear All
Columns
Analyze
New Analysis
Stop

98 Documents

No.	Title	URL	Source
1	DASADA Program I...	http://schaferc...	AllTheWeb msn ...
2	DASADA Project	http://www.isr....	AllTheWeb Goog...
3	DASADA project h...	http://www.cs....	AllTheWeb Goog...
4	AFRL-Rome DASA...	http://www.rl.a...	msn HOTBOT
5	Columbia Universit...	http://www.psl...	AllTheWeb HOT...
6	DASADA Program ...	http://schaferc...	AllTheWeb msn
7	rann_kutch	http://www.ah...	EXCITE HOTBOT
8	dasada page	http://dasada.p...	AllTheWeb Google
9	Salt ends Kutch far...	http://www.ex...	AltaVista EXCITE
10	TBASSCO -- Relat...	http://www.isi....	msn HOTBOT
11	BBN-DASADA &qu...	http://aai.bbn.c...	AllTheWeb Google
12	DASADA	http://schaferc...	AllTheWeb Google
13	Software Engineeri...	http://www.cs....	AllTheWeb Google
14	AFRL-Rome DASA...	http://www.if.a...	EXCITE
15	AFRL-Rome DASA...	http://www.rl.a...	Google

Dasada

Type: Search Engin...

✓	altavista	Auto...
✓	Google!	Goo...
✓	altavista	AltaV...
✓	excite	EXCI...

Result size: 20

● Category ○ Business Search

✓ Merge search results

View 'Dasada' is added.

Figure 5.2: The GeoWorlds Client and the Information Manager are the primary user interface for developing incidents and gathering data from web-based search utilities.

5. Gauges must be developed.

The GeoWorlds system is launched by running a series of perl scripts. In order for the Active Interfaces Probe Run-Time Infrastructure to monitor GeoWorlds, they must be started in the same Java Virtual Machine. To accomplish this we modified the GeoWorlds launching perl scripts so that they launch GeoWorlds in the AIPRI Shell. The change was simple, requiring the alteration of only two lines of the script. The first line was to include in the classpath the AIPRI classes and to prepend the location of the probed classes to. The second line launches the AIPRI shell with the GeoWorlds main class as an argument.

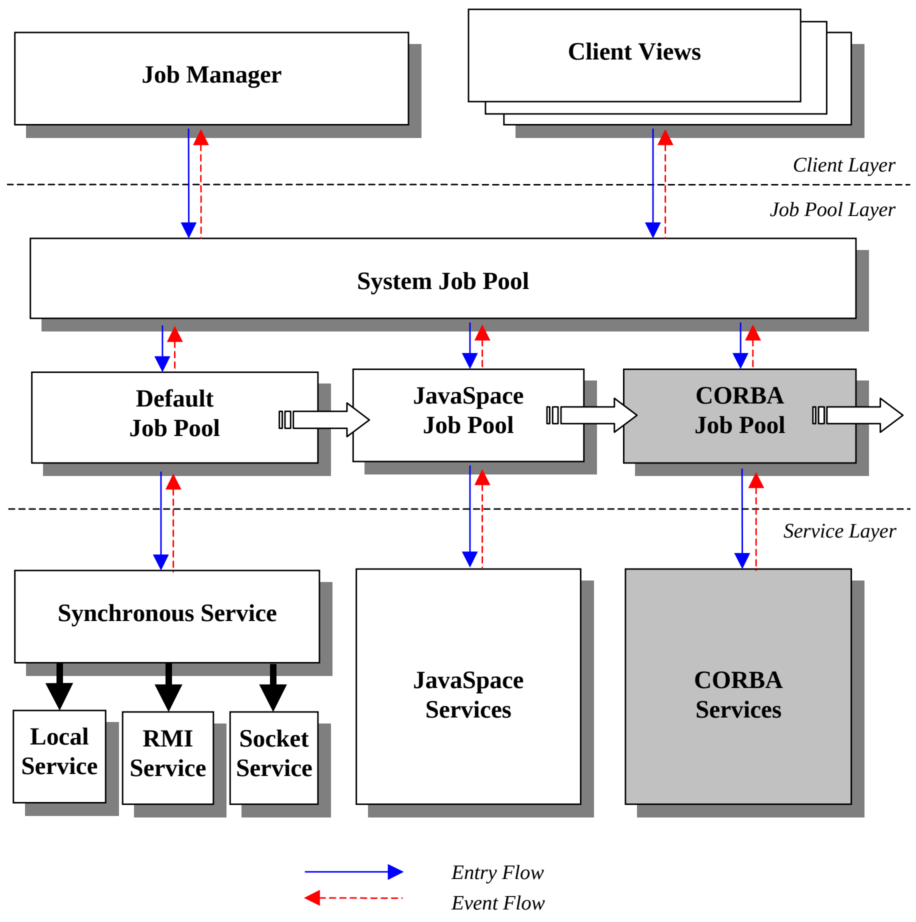


Figure 5.3: The GeoWorlds Dasher Service and Job Management Architecture oversees the use of the distributed (and web-based) services on which GeoWorlds Relies.

The AIPRI shell would then launch the probe adapter and GeoWorlds.

Perhaps the most important step is developing an understanding of how the system works and identifying target classes. This is critical to designing useful gauges and the correct placement of probes to gather the necessary information. In general it is expected that people familiar with the system would identify probe locations. The actual GeoWorlds classes that were probed will be described in the next section.

The next step in preparing GeoWorlds for monitoring with Active Interfaces was the insertion of the active interface hooks. The AIDE compiler was used to insert hooks into the classes and prepare them for use in the AIPRI.

Beyond these preparation stages it is only necessary to develop and deploy the probes and gauges that will be used for the monitoring. These steps will be described in the next section.

5.7 GeoWorlds Gauges and Demonstrations

KX was demonstrated at the DARPA DASADA Demo Days conference in Baltimore Maryland on June 4-6, 2001. We demonstrated the Active Interface Development Environment and presented two gauges that we developed show Active Interfaces and the Active Interface Run-Time Infrastructure in action. In addition to being used in our demonstration, Active Interfaces and the AIPRI were used by Columbia University as the probe infrastructure for the development of their demonstration. In the remainder of this chapter we present the gauges and probes used to for our demonstration.

5.7.1 QueryWatcher

As an early test of the AIPRI's ability to monitor GeoWorlds we developed `QueryWatcher`. `QueryWatcher` is an Active Interface enabled probe that monitors web queries generated by GeoWorlds users. `QueryWatcher` maintains a list of trigger words. If the user performs a query that contains one of the trigger words, `QueryWatcher` generates an event. The `QueryWatcher` focus method provides a mechanism for the run-time addition or removal of trigger words. A gauge was created that tracked the incidence of trigger word queries. `QueryWatcher` was built to demonstrate the possible utility of having access to method parameters from the target system. To monitor the queries it was necessary to extract the information from objects passed as parameters to the `SystemJobPool`.

5.7.2 EBEs

One of the main information sources used by GeoWorlds is data returned by web-based search engines. GeoWorlds allows a user to query multiple search engines at a time. The way the GeoWorlds service architecture is constructed no results are returned to the user until all of the search engines return. The results are then merged and displayed all at once.

GeoWorlds does not provide any feedback on the performance of the various sites being used. When users experience slow search times they have no idea what site is to blame. To overcome this we developed an experience based expectation gauge to monitor the performance of the web-based search facilities used by GeoWorlds.

The bar graph gauge in figure 5.4 was used to track search performance by the search engines used. In addition to displaying the performance of the current query, the gauge shows the fastest, slowest and average search speeds for each of the sites. The GeoWorlds application allows us to exclude hosts from query processing. Feedback from our gauge allowed us to identify slow hosts. We then used GeoWorlds GUI to eliminate the slow sites from future searches.

To generate the appropriate information we probed two classes from GeoWorlds service launching architecture:

- `edu.isi.dasher.webwrapper.WebWrapperServer$RunWebWrapper`
- `edu.isi.dasher.webwrapper.ExtractWebLinks`

We probed before and after the `run` method of the `RunWebWrapper` class to get the time taken for the general case when results are returned by the search engine. To calculate the time we get the current time when the before callback is executed. We again get the current time when the after callback is executed. The difference is the time that the service was executing. Since the service architecture is multi-threaded we

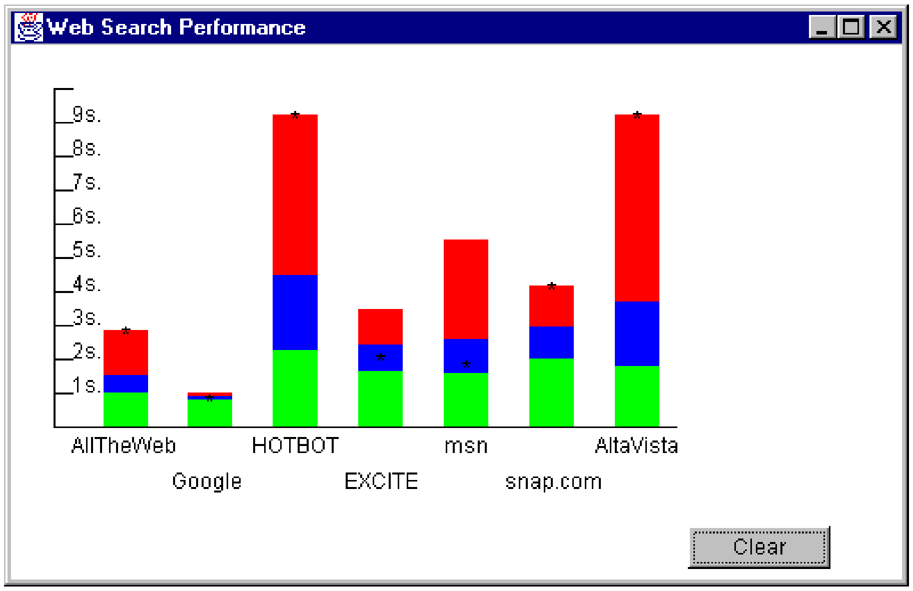
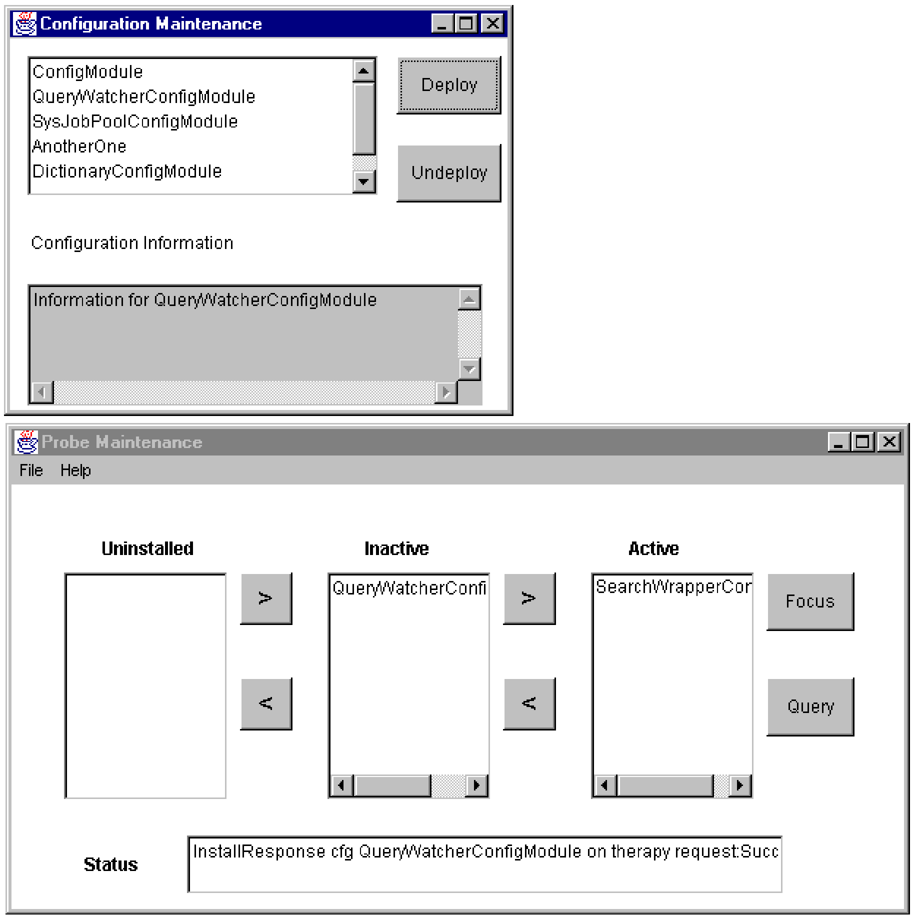


Figure 5.4: Search Engine Performance Gauge. This gauge shows the current, fastest, slowest and average performance of the search engines utilized by GeoWorlds.

need a way of identifying which before time correlates to which after time. We use the static `Thread.currentThread().getName()` methods to return the currently executing threads name. We use this name as the key on which we hash the before time. When we get after callbacks we again retrieve the thread name and use it extract the before time. We also probed the `quit` method of the `RunWebWrapper` class so that we could terminate nicely if the service times out. The `RunWebWrapper.run` method is concerned with the low level details of executing the queries. It is not possible to extract the search engine name from the callbacks on the run method. Therefore in addition to probing `RunWebWrapper`, we probed the `getLinks` method of the ExtractWebLinks class. From the arguments to the `getLinks` method we are able to identify the search engine for which the current query is being run. To match the search engine name with the time we again utilize the `Thread.currentThread().getName()` method to extract the name of the currently executing `thread` and correlate the search engine name with the running times extracted from the `RunWebWrapper` class.

5.7.3 Upgraded Probe Launcher

For the DASADA Demo Days we developed an upgraded Probe Launcher (see Figure 5.5). The new probe launcher shows the state of the various probe configurations on the target system. It also maintains persistent information that allows it to resynchronize with the target system after a restart.



ConfigModule
QueryWatcherConfigModule
SysJobPoolConfigModule
AnotherOne
DictionaryConfigModule

Configuration Information

Information for QueryWatcherConfigModule

QueryWatcherConfi

SearchWrapperCor

InstallResponse cfg QueryWatcherConfigModule on therapy request:Succ

Figure 5.5: The GeoWorlds Demonstration Probe Launcher.

5.8 Conclusion

In this chapter we described the integration of the AIPRI with KX, the monitoring system developed collaboratively with Columbia University. The monitoring system was shown at the DASADA Demo Days conference and served as a solid proof of concept for the idea of a generic monitoring infrastructure.

Chapter 6

Conclusions

In this thesis we made contributions in three main areas.

The first contribution was related to probes. We examined the ways that probes can collect information from running systems and cataloged a number of existing technologies that can serve as probes. We explicitly described a number of probe types that can be implemented using Active Interfaces.

The second contribution was the development of a Probe Run-Time Infrastructure that can be used to deploy and manage probes at run time. We described a high level design that would allow for implementation of the run-time infrastructure for the various probe mechanisms. We provided an implementation and demonstration of the design for Active Interfaces.

The third contribution of this thesis was participation in the development of a demonstration of the full monitoring architecture to monitor GeoWorlds. This demonstration was shown at the DASADA Demo Days conference. We utilized the Active Interfaces Probe Run-Time Infrastructure to support the development of probes to collect information from GeoWorlds at run-time. Additionally we assisted in the development of gauges and meaning full scenarios which highlight the ability of the monitoring system to detect problem in the system.

6.1 Future Work

There are a number of areas where work can continue on the implementation of the Active Interface Probe Run-Time Infrastructure. The current implementation does not take into account the possible presence of multiple probe controlling entities in the monitoring system. Any entity can shutdown a probe, with out regard to its use by other entities. Siena's subscription mechanism provides a way that parties that are interested only in certain events from a probe can narrow their subscription. However, for efficiency purposes, it is undesirable to have all of the filtering take place in sienna. This limitation can be overcome by integrating some support for tracking interested parties into the Probe Infrastructure. If no interested parties exist for a particular event it will not be delivered into Siena. Additionally, implementations of the Probe Run-Time Infrastructure could be built for the other probing mechanisms defined here.

This was the first year of a multi-year project. As such it was designed as a proof of concept and has not yet reached its full potential. One of the longer term goals of the DASADA project is to allow checking of the correctness of software systems at run-time. In this first year we have concentrated on information collection and dissemination. We built gauges that measured properties of the system that are easy to quantify. In future work the types of gauges implemented should be expanded to include poset evaluation gauges that can determine if the communication patterns of an individual components or collection of components match previously defined partially ordered sets of events. A major requirement for the implementation of this type of gauge is deep understanding of the system being monitored.

Appendix A

Probe Runtime Infrastructure APIs

All APIs are described as events so that probes can be remotely controlled and the data they produce remotely consumed. The syntax shown below is for publishing the event. Each published event is received by those clients who register interest in that event. Each Probe event is acknowledged with a partner event containing the status of the operation (in most cases, success or failure).

A.1 Core Probe Management API

These events are delivered to Siena from Management interfaces designed to deploy, install, and activate probe configurations. It is possible that gauges or gauge managers (see Gauge Infrastructure working group) may also issue these managerial events.

1. Deploy a Probe Configuration (event type: deploy, response: deployResponse)

```
Deploy (in String Probe-Configuration-Name, in String Hostname,
         in URL Probe-Configuration-Module)
```

> The Probe-Configuration-Module defining the named Probe-Configuration-Name becomes an available probe configuration on the machine identified by Hostname. The Probe-Configuration-Module is a URL that packages the code and declarations needed to construct instances of the probe configuration. The format of the probe package is language-specific but otherwise shall be a generic standard, such as Java Archive, Tape Archive, or a Zip file.

2. Install a Probe Configuration (event type: install, response: installResponse)

```
Install (in String Probe-Configuration-Name, in String Hostname,
     in String TargetSystem)
```

The already deployed Probe-Configuration-Name on Hostname is incorporated into the named running TargetSystem. The probes defined in the probe configuration are initialized to their deactivated state (i.e. not sensing any behavior in the running system). If TargetSystem is not running at the time the Install event is received, then Probe-Configuration-Name is installed when TargetSystem is started on Hostname (this enables probes to sense startup behavior and corresponds to how statically placed probes would be deployed and installed). The Probe Adaptor (see Section) is responsible for instantiating or constructing probes when the probe configuration is installed on a running system.

3. Activate a Probe Configuration (event type: activate, response: activateResponse)

```
Activate (in String Probe-Configuration-Name, in String Hostname,
              in String TargetSystem)
```

The already installed Probe-Configuration-Name on TargetSystem on Hostname is activated so that it senses behavior in the running system. In this activated state, probes may issue Sense events for some subset of the behavior they observe. Note: If an Activation event is received before the named TargetSystem is running, then Probe-Configuration-Name is activated when TargetSystem is initially started on Host.

4. Undeploy a Probe Configuration (event type: undeploy, response: undeployResponse)

```
Undeploy (in String Probe-Configuration-Name, in String Hostname)
```

Resources for the already deployed Probe-Configuration-Name on TargetSystem on Hostname are released. Note: If an Undeploy event is received while TargetSystem is not running, then the specified probe configuration is deactivated and uninstalled as appropriate prior to being undeployed.

5. Uninstall a Probe Configuration (event type: uninstall, response: uninstallResponse)

```
Uninstall (in String Probe-Configuration-Name, in String Hostname,
              in String TargetSystem)
```

The specified Probe-Configuration-Name on TargetSystem on Hostname is uninstalled. The probe may be removed from the Target System. Note: If an Uninstall event is received while TargetSystem is not running, then the specified probe configuration is deactivated, as appropriate, prior to being uninstalled. An uninstalled probe must be reinstalled before being activated.

6. Deactivate a Probe Configuration (event type: deactivate, response: deactivateResponse)

```
Deactivate (in String Probe-Configuration-Name, in String Hostname,
              in String TargetSystem)
```

The specified Probe-Configuration-Name on TargetSystem on Hostname is deactivated. The probe will cease to emit sensed events. Note: If a Deactivate event is received while TargetSystem is not running, then the specified probe configuration is deactivated.

A.2 Advanced (Optional) Probe Management API

Because there are a wide variety of probes, there are additional events that a probe may be able to support. These are initially defined as Advanced, or Optional, probe responsibilities. In future versions of the API, these may be reclassified as Core Probe Management.

1. Query a probe for the events it can generate (event type: querySensed, response: generateSensed)

```
Query-Sensed (in String Probe-Configuration-Name, in String Hostname)
```

Requests a list of all of the Event-Names that the Probe-Configuration-Name can generate while it is activated. This request is answered through a generateSensed event. Note: A probe must be deployed to be capable of responding to this event. If the desired probe is not installed, the Probe Adaptor is responsible for extracting this information from the packaged probe information.

2. Reconfigure a probe configuration (event type: focus, response: focuseResponse)

```
Focus (in String Probe-Configuration-Name, in String Hostname,
         in String TargetSystem, in StringPairVector focusData)
```

focusData contains a vector of pairs of the form ¡parameter, value¿ that are passed to the already installed Probe-Configuration-Name on TargetSystem on Hostname. The purpose of this event is to enable a probe to "focus" its sensors as specified by the parameters. How the probe interprets these parameters is entirely probe configuration specific. The desired probe must be installed for the focus event to be processed.

A.3 Events from Probes to Probe Infrastructure

There are many events that simply contain the status of a probe monitoring request, including: deployResponse, installResponse, activateResponse, undeployResponse, uninstallResponse, deactivateResponse, and focusResponse. In this section we describe in more detail only those events that contain additional non-obvious information.

1. To respond to Query Sensed events: (event type: generateSensed)

```
Generate-Sensed (in String Probe-Configuration-Name, in String Hostname,
                 in Vector EventNames)
```

This event is issued in response to a querySensed event for the list of all Event-Names that the named Probe-Configuration-Name can generate when activated. This list is returned as the EventNames vector.

2. An activated probe emits Sensed events as it monitors the target system (event type: sensed)

```
Sensed (in String Probe-Configuration-Name, in IntervalType Event-Type,
                 in StringPairVector SensedValues)
```

Probes in the activated state may issue sensed events that identify some subset of the behavior they observe. The Probe-Configuration-Name identifies the probe that emitted the sensed event. Event-Type is either Start, Point, or End which specifies, respectively, the start of an event interval, the occurrence of a point event (i.e,. an event with no duration), or the end of an event interval. SensedValues is a Vector of String pairs ¡attribute, value¿ to reflect probe-specific information extracted from the target system. Note: It is not mandated that a probe emit an event that identifies the TargetSystem or the Hostname of the computer on which it executes. The probe can include such information in SensedValues using the attribute names "TargetSystem" and "Hostname".

Bibliography

[1] Anant Agarwal, Richard Sites, and Mark Horwitz. Atum: A new technique for capturing address traces using microcode. In *Proceedings of the 13th International Symposium on Computer Architecture*, June 1996.

[2] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[3] Robert Balzer. Draft probe run-time infrastructure. E-mail compilation of discussions with DASADA Collaborators, January 22 2001.

[4] Robert Balzer and Neil Goldman. Mediating connectors. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware*, pages 73–77, 1999.

[5] Antonio Carzaniga, Elisabetta Di Nitto, David Rosenblum, and Alexander Wolf. Issues in supporting an event based architectural style. In *Proceedings of the Third International Software Architecture Workshop*, Beuna Vista, Florida, November 1998.

[6] Antonio Carzaniga, David Rosenblum, and Alexander Wolf. Interfaces and algorithms for a wide-area event notification service. Technical Report CU-CS-888-99, University of Colorado, Department of Computer Science, October 1999. Revised May 2000.

[7] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, 2000.

[8] Geoff Cohen, Jeff Chase, and David Kaminsky. Automatic program transformation with joie. In *Proceedings of the 1998 USENIX Annual Technical Symposium*, 1998.

[9] Murilo Coutinho, Robert Neches, Ke-Thia Yao Alejandro Bugacov, Vished Kumar, In-Young Ko, Ragy Eleish, and Sameer Abhinka. Geoworlds: A geographically based information system for situation understanding and management. In *Proceedings of the First International Workshop on TeleGeoProcessing (TeleGeo '99)*, Lyon, France, May 1999.

[10] Markus Dahm. Byte code engineering. In *Proceedings JIT'99 (Java-Informations-Tage)*, September 1999.

[11] DARPA DASASA Project Homepage:
`http://schafercorp-ballston.com/dasada/index2.html`, 2000.

[12] Enrico Buonanno Proposal for Work on the Event Distiller
`http://canal.psl.cs.columbia.edu/bscw/bscw.cgi/d8262/enrico.htm`, 2001.

[13] Susan Eggers, David Keppel, Eric Koldinger, and Henry Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 8, May 1990.

[14] An Invitation to Eiffel
`http://www.eiffel.com/doc/manuals/language/intro` , 2000.

[15] Erich Gamma, Richard Helm, Ralf Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., 1995.

[16] Phil Gross. *Smart Event Schemas*. Columbia University Programming Systems Laboratory, 0.1 edition, January 2001. `http://canal.psl.cs.columbia.edu/bscw/bscw.cgi/0/6216`.

[17] George Heineman and Alok Mehta. Architectural evolution of legacy systems. In *23rd Annual International Computer Science and Application Conference (COMPSAC-99)*, February 1999.

[18] George T. Heineman. A model for designing adaptable software components. In *22nd Annual International Computer Science and Application Conference (COMPSAC-98)*, pages 121–127, Vienna, Austria, August 1998.

[19] Urs Hoelzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer Verlag, 1993.

[20] Gail Kaiser and George Heineman. Coping with complexity: A standards-based kinesthetic approach to monitoring non-standard component-based systems. Technical Proposal for DARPA's DASADA program, May 2000.

[21] Ralph Keller and Urs Holzle. Binary component adaptation. Technical Report TRCS97-20, University of California, Santa Barbara, December 1997.

[22] Ralph Keller and Urs Holzle. Supporting the integrations and evolution of components through binary component adaptation. Technical Report TRCS97-15, University of California, Santa Barbara, September 1997.

[23] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software, Practice and Experience*, 24(2), 1994.

[24] Han Bok Lee. Bit bytecode instrumenting tool. Master's thesis, University of Colorado, 1997.

[25] Han Bok Lee and Benjamin G. Zorn. Bit: Bytecode instrumenting tool. In *USENIX Symposium on Internet Technologies and Systems*, December 1997.

[26] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[27] Nenad Medvidovic and Richard Taylor. Separating fact from fiction in software architecture. In Jeff N. Magee and Dewayne E. Perry, editors, *Third International Workshop on Software Architecture*, pages 105–108, Orlando, Florida, November 1998.

[28] Alexandre Santoro, Walter Mann, Neel Madhav, and David C. Luckham. ejava extending java with causality. In *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering*, June 1998.

[29] Mary Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first class status. In *Proceedings of the Workshop on Studies of Software Design*, 1993.

[30] Siena API documentation
`http://www.cs.colorado.edu/serl/siena/software/java/index.html`,
2001.

[31] Smart Event Schema
`http://canal.psl.cs.columbia.edu/bscw/bscw.cgi/0/5191`,
2001.

[32] *Unix Man pages for prof, gprof and pixie*, solaris8 edition.

[33] Amitabh Srivastava and Alan Eustace. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[34] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized probram analysis tools. In *Proceedings of SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, June 1994.

[35] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., and Jason E. Robbins. A component- and message-based architectural style for gui software. In *Proceedings of the Seventeenth International Conference on Software Engineering (ICSE17)*, pages 294–304, Seatle, Washington, April 1995.

[36] David W. Wall. Systems for late code modification. Technical Report 92/3, WRL, 1992.