# SONAR: System Optimization and Navigation with Aspects at Runtime

Chunjian Robin Liu
University of Victoria
cliu@cs.uvic.ca

Celina Gibbs
University of Victoria
celinag@cs.uvic.ca

Yvonne Coady
University of Victoria
ycoady@cs.uvic.ca

## ABSTRACT

Traditional system optimization and navigation techniques, based on static system structure and static instrumentation, are not suitable for understanding and handling today's complex, distributed and dynamic systems at runtime. This paper introduces an approach we call *SONAR* (System Optimization and Navigation with Aspects at Runtime). Through a combination of Extensible Markup Language (XML), dynamic Aspect-Oriented Programming (AOP) and Java Management Extensions (JMX), SONAR provides a fluid and unified framework for runtime system optimization and navigation. In particular, dynamic aspects are used to integrate scattered code level artifacts across predefined abstraction boundaries – application, framework/middleware/virtual machine, operating system – together as a meaningful entity with respect to users' interests. Such aspects can be defined, enabled and disabled by a stakeholder at runtime. A preliminary evaluation of SONAR shows only minimum impact on performance and a relatively small memory requirement for large systems.

## 1. INTRODUCTION

Today's systems are increasingly challenging to holistically analyze due to both their size and complexity. Though layering, componentization, and virtualization can be leveraged to provide meaningful abstractions at various levels, these abstractions also make system-wide navigation and analysis, which must cross these predefined boundaries, more difficult than in simple flat systems. For example, to perform a root-cause fault-analysis, exceptions and states at various levels of the software stack need to be collected and analyzed collectively. This is difficult because application level exceptions are usually handled by the middleware and hidden from both the lower layers and the users. Likewise, certain exceptions at lower levels can be hidden or transformed to a different representation for higher levels to digest [2,3]. To be able to understand the root-cause of a problem, or some other systematic behaviour, information from entities across layers and/or distributed at different locations needs to be easily collected and correlated.

Looking at this problem from another angle, today's complex system architectures are designed and documented with multiple views (structural - class, deployment; behavioral - use case, sequence, collaboration; control flow, data flow) for multiple stakeholders. At an implementation stage, these design views are mapped into code level artifacts (module/component, file, package, class, method). For structural views, the mapping is usually direct and explicit, since current object-oriented programming methodology provides a hierarchical/structural decomposition and code is developed based on such decomposition. Other views, such as behaviour views, are typically difficult to map or realize from hierarchical/structural decomposed code level artifacts. Furthermore, users with different roles, i.e. designer, developer, administrator and maintainer, might have different interests and therefore require different views of a single system at runtime. These views may be directly derived from design views, or may represent new perspectives based on more specific stakeholder interests.

In addition to these increasing demands, increasing heterogeneity of large systems, dynamic features of programming languages, frameworks/middleware/virtual machines and operating systems, and other advanced techniques (such as configuration, adaptation and autonomic computing) make the overall system more dynamic and as a result, static instrumentation techniques are not suitable for navigation and analysis such systems. For example, by using Java reflection, a level of indirection is added and therefore a program can achieve higher degree of flexibility. However, it also makes realizing which object is actually invoked more difficult than a static call. For systems such as pervasive systems, which feature a high degree of dynamism in terms of device/service states and availability, static based techniques are equally unsuitable for analyzing runtime behavior.

In order to understand system-wide runtime behavior, certain entities in the system (or even from the outside environment) might need to be correlated together and represented as a single entity – coarser/finer grained or even crosscutting – for navigation and other tasks. Moreover, such views should be able to be reconstructed when users' interests change. Runtime analysis of a problem has been shown to typically be an iterative process, with an analyst's focus constantly changing during the process of analyzing [4]. Thus, views need to be easily defined and constructed dynamically across the predefined boundaries of class, component, subsystem, layer and so on. Ideally, views should be able to be easily removed once users no longer need them, and further incur little to no performance penalty. Simply put, systems need to be abstracted and viewed differently and dynamically.

Based on this premise, SONAR was motivated by this need for a fluid mechanism to dynamically integrate all scattered artifacts (entities and data) cross predefined vertical and/or horizontal abstraction boundaries together to represent a meaningful entity

with respect to a user's interest. As a result, SONAR was designed with three key goals in mind:

1. **Language/framework agnostic definition:** As there are now an increasing number of AOP frameworks for different programming languages. SONAR can be configured to leverage this variety in order to diagnose more systems.

2. **Dynamic instrumentation:** Instrumentation code should be able to be inserted dynamically. Furthermore, such code should be able to be removed with zero impact.

3. **Simple visualization and management:** Collected data should be easily visualized and managed. Furthermore, management should also be standard-compliant.

The rest of this paper is organized as follows: Section 2 covers some background information. Section 3 explains SONAR's architecture and describes relevant implementation details showing how some of the most challenging goals above are accomplished. In Section 4, we evaluate our implementation based on performance, memory footprint, and other factors. Finally, we discuss related work and future research directions in Section 5 and conclude in Section 6.

## 2. Background

This section briefly introduces the three key technologies used by SONAR – XML, AOP, and JMX – and the specific ways in which SONAR uses each.

### 2.1 Why XML?

Extensible Markup Language (XML) was originally designed to improve the functionality of the Web by providing more flexible and adaptable information identification[1]. Given that XML is not a fixed format like HTML (a single, predefined markup language), it can be generally used to customize markup languages.

SONAR uses XML's ability to encapsulate information in order to pass it between different computing systems which would otherwise be unable to communicate.

### 2.2 Why AOP?

Aspect-oriented programming (AOP) modularizes crosscutting concerns – concerns that are present in more than one module, and cannot be better modularized through traditional means [7]. Looking at an aspect, a developer can see both the internal structure of a crosscutting concern, and its interaction with the rest of the program during execution. Three key elements of AOP are *joinpoints, pointcuts* and *advice.* Joinpoints are well-defined points in the program flow such as method call/execution, and

field accesses, pointcuts select certain execution points in the program flow, and advice provides code to run at these selected points. Dynamic AOP allows aspects to be introduced/removed to/from a system at runtime.

SONAR leverages dynamic AOP's ability to structure system-wide crosscutting concerns for dynamic analysis, and introduce/remove them at runtime.

### 2.3 Why JMX?

Originally known as Sun's *JMAPI*, Java Management Extensions (JMX) [5] is gaining momentum as an underlying architecture for J2EE servers. It defines the architecture, design patterns, interfaces, and services for application and network management and monitoring. Managed beans (MBeans) act as wrappers, providing localized management for applications, components, or resources in a distributed setting. MBean servers are a registry for MBeans, exposing interfaces for local/remote management. An MBean server is lightweight, and parts of a server infrastructure are implemented as MBeans.

SONAR leverages JMX's support for highly modular and customizable server architectures, and standard support for management features.

## 3. SONAR Design and Implementation

The subsections that follow provide an overview of SONAR's current design and implementation. Further features and enhancements are discussed in Section 6.

### 3.1 SONAR Architecture

In order to address the goals described in Section 1, the overall architecture of SONAR is organized as shown in Figure 1. Aspects are generated from XML-based definition files, deployed to instrumented applications, application framework/middleware, and virtual machine. They are then managed through JMX compatible tools through domain independent API such as those related to deployment/undeployment of aspects and domain specific API such as system navigation. SONAR is thus designed and implemented in three main parts: AOP integration, XML transformation/code generation and JMX management. Each corresponds to one goal in Section 1, and is further explained in detail in the following subsections.
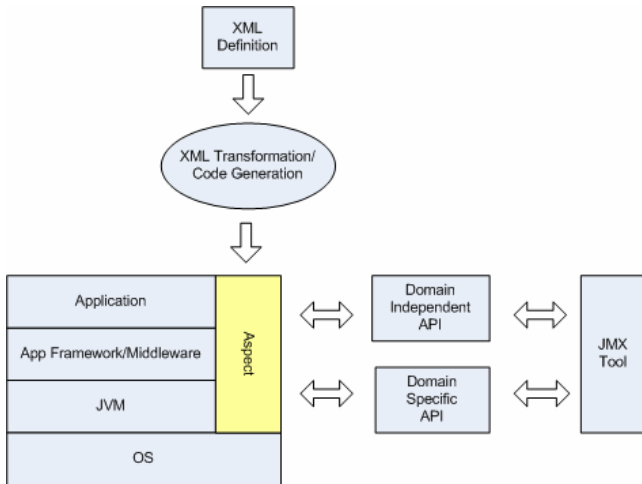
---

[1] http://www.ucc.ie/xml/#acro

**Figure 1. SONAR overall architecture.**

## 3.2  AOP Integration

To achieve dynamic instrumentation, we chose dynamic AOP since it provides a language level (code centric) support for augmenting existing systems for various purposes. AOP's rich joinpoint model provides a solid foundation for implementing instrumentation. The joinpoint model covers almost all execution points in a system written in certain languages. These points include method invocation/execution, field access and so on. This enables fine-grained instrumentation – almost all significant events in the source code can potentially be instrumented.

Dynamic AOP further provides a powerful mechanism for runtime aspect manipulation such as runtime deployment/undeployment. In other words, advice can be dynamically woven into targets and dynamically removed from targets. Our implementation is based on a Dynamic AOP implementation, AspectWerkz[1].
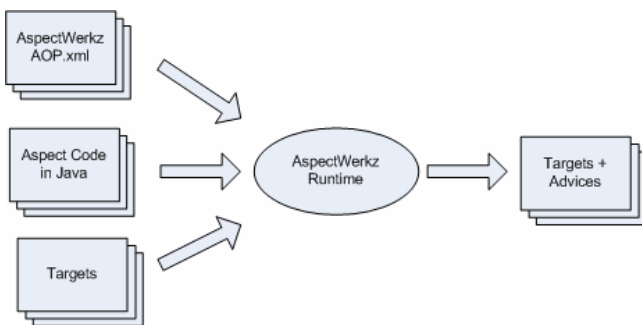


**Figure 2. Illustrates how configuration file, aspects and targets are used in AspectWerkz.**

## 3.3  XML Transformation/Code generation

### 3.3.1  XML Definition

Aspects are defined in AOP framework independent XML files. Therefore, they can be implemented using different AOP frameworks or even in different programming languages such as Java, C++.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <sonar>
3    <system name="test" start="auto">
4      <aspect name="httpMonitor" class="sonar.aspect.MonitorAspect"
         deployment-model="perJVM" manageable="true" target-language="java">
5        <pointcut name="methodToMonitor" expression="execution(*
           org.apache.coyote.http11.Http11Processor.process(..))"/>
6        <advice name="monitorTest(JoinPoint)" type="around"
           bind-to="methodToMonitor">
7          <action language="java" domain="trace" type="before">
8            <![CDATA[
9              log("...");
10           ]]>
11         </action>
12         <action language="java" domain="trace" type="after">
13           <![CDATA[
14             log("...");
15           ]]>
16         </action>
17       </advice>
18       <param name="..." value="..."/>
19     </aspect>
20   </system>
21  </sonar>
```

**Figure 3. Sample XML definition file.**

The core content of an aspect includes: variable/method declaration, pointcut expression, advice with actions, parameter definition. Variable/method declarations and actions are discussed in details in the next 2 sections. The current schema is mainly based on AspectWerkz's aspect definition schema, which is very similar to many other existing AOP frameworks, with additions required for transformation and code generation. Aspects can be declared to be started automatically or manually. Automatically started aspects are enabled when the target systems are loaded, while manually started aspects have to be explicitly manually enabled at the runtime.

### 3.3.2  Domain Specific API or Language

The target domain in SONAR prototype implementation is system optimization and navigation. Variable/method declarations and actions contain code targeting this specific domain. The reason for using a domain specific API or language is to separate aspect code from domain specific implementation. For example, the log method used in the Figure 3 is defined outside of the aspect code. It can be implemented as printing to screen, writing to a log or sending to some management console. As a result, the implementation choice of such method can be made independently from aspect code and therefore, can be adjusted based on the target system. Other reasons are to ease the development and limit the impact of aspects. In the current implementation, declarations and actions are written only in Java.

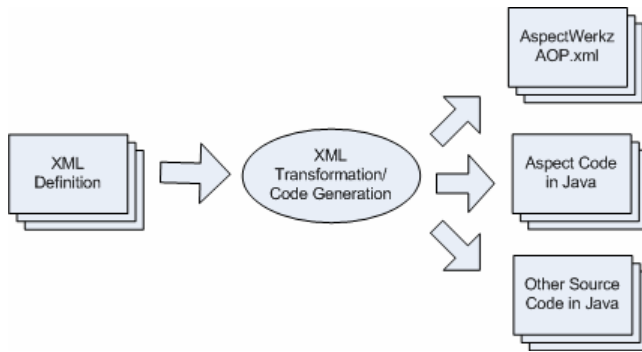### 3.3.3  XSLT Transformation & Code Generation

**Figure 4. XML definition files are transformed into other XML files and source code in target language using XSLT and domain specific compiler/code generator.**

As shown in Figure 4, XSL Transformation (XSLT) is used to transform XML definition files into other XML files, such as the aspect definition file for AspectWerkz (Figure 5), aspect code (Figure 6) and other necessary source code such as interfaces and helper classes for management purposes. If variable/method declarations and actions in the aspect definition are written in domain specific languages, domain specific compiler might be used to compile the code into the language used in the target system.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3    <!DOCTYPE aspectwerkz PUBLIC "-//AspectWerkz//DTD//EN"
     "http://aspectwerkz.codehaus.org/dtd/aspectwerkz2.dtd">
4
5    <aspectwerkz>
6      <system id="test">
7        <aspect name="httpMonitor" class="sonar.aspect.MonitorAspect"
         deployment-model="perJVM">
8          <pointcut name="methodToMonitor" expression="execution(*
           org.apache.coyote.http11.Http11Processor.process(..))"/>
9          <advice name="monitorTest(JoinPoint)" type="around"
           bind-to="methodToMonitor"/>
10         <param name="..." value="..."></param>
11     </aspect>
12   </system>
13 </aspectwerkz>
```

**Figure 5. AspectWerkz's aop.xml generated by transforming the definition file in Figure 3.**

```
1  package sonar.aspect;
2
3  import org.codehaus.aspectwerkz.*;
4  import org.codehaus.aspectwerkz.definition.*;
5  import org.codehaus.aspectwerkz.joinpoint.*;
6  import org.codehaus.aspectwerkz.transform.inlining.deployer.*;
7
8  import sonar.util.*;
9
10 import java.lang.management.*;
11 import javax.management.*;
12 import javax.management.openmbean.*;
13
14 public class MonitorAspect implements MonitorAspectMBean {
15   private final AspectContext aspectContext;
16
17   public MonitorAspect(final AspectContext aspectContext) {
18     this.aspectContext = aspectContext;
19   }
20
21
22   public Object monitorTest(final JoinPoint joinPoint) throws Throwable {
23     log("...");
24
25     final Object result = joinPoint.proceed();
26
27     log("...");
28
29     return result;
30   }
31 }
```

**Figure 6. Java source code containing AspectWerkz specific code generated from the definition file in Figure 3.**

## 3.4 JMX Management

JMX is used as a means to manage the aspects. This includes retrieving data from aspects, invoking operations and receiving event notification. Through JMX, the aspects can be managed by JMX-compatible tools remotely and/or locally. The tool we used is called JConsole which is a JMX-compliant graphical tool for monitoring and management and is built in Sun's JDK distribution. Figure 7 and Figure 8 show how JConsole can be used to manage aspects, more specifically MonitorAspect as implemented in SONAR. MonitorAspect monitors HTTP request, database access and JSP service. The Figure 7 illustrates how the invocation statistics collected by MonitorAspect from those three different instrumentation points are visualized as line charts in JConsole. These data and therefore charts are automatically updated. Figure 8 shows the operations supported by MonitorAspect. The deploy()/undeploy() are used to deploy/undeploy MonitorAspect. After being undeployed, advices defined in MonitorAspect are removed from their targets. MonitorAspect still can be accessed by JConsole. Therefore, it can be redeployed by invoking deploy() from JConsole.

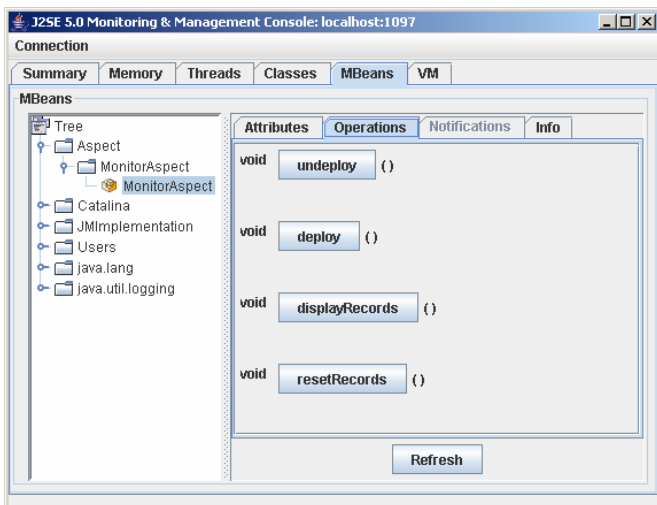**Figure 7. Data from aspects is visualized and can be updated in JConsole.**



**Figure 8. Operations of aspects are listed and can be invoked in JConsole.**

## 4. ANALYSIS/EVALUATION

Currently, there is no JVM that supports schema redefinition of any loaded classes. In other words, changes (such as add, remove or rename fields or methods, change the signatures of methods, or change inheritance) are not allowed at runtime[2][1]. However, dynamic deployment/undeployment of aspects in AspectWerkz requires schema changes to target classes.

To get around with the above restriction, AspectWerkz uses a preparation mechanism to prepare target classes for later

---

[2]  As mentioned in Java API's instrumentation section, such restriction might be lifted in the future.

deployment/undeployment at runtime. The special construct, named deployment scope, is used to specify the joinpoints to be prepared – by adding a call to a public static final method that redirects to the target join point. The added indirection will surely introduce overhead. However, such indirection can be inlined by most modern JVMs [1].

As a result, it is not possible to achieve absolute zero impact on the performance and memory footprint on target classes. Table 1 shows the impact of using AspectWerkz, with regarding to class file size and runtime performance. The added size to target class file is around 1000 bytes.

**Table 1. Impact on file size and performance.**

|  | Size (bytes) | Runtime Performance (ms) |
|---|---|---|
| **Original** | 1,064 | 0.281 |
| **After preparation (no advice weaven)** | 2,447 | 0.282 |
| **After weaven** | 1,839 + the size of the aspect(s) | 0.328 |

As indicated in the Table 2, Tomcat server's startup time is significantly increased because it is running under AspectWerkz' online mode – aspects are weaved into target classes when they are loaded into JVM.

**Table 2. Startup time of Tomcat. Sampled by running Jakarta Tomcat 5.5.4. Spring Framework 1.1.3, Spring JPetstore and AspectWerkz 2.0 (RC2) with online mode on JDK 1.5.0.**

| Configuration | Start up time (ms) |
|---|---|
| **Tomcat** | 5,953 |
| **Tomcat with JMX enabled** | 6,122 |
| **Tomcate with AspectWerkz and JMX enabled** | 27,883 |

One limitation of the current joinpoint model used in defining aspects in SONAR's XML files is that it is based on the joinpoint model from AspectJ and AspectWerkz. Thus, it is closely targeted to object-oriented programming language model. Many advanced features are not available to systems developed in non-object oriented languages. This makes integration with legacy systems harder. Moreover, AOP provides a low-level code centric approach. It is focused on static structure of code such as control flow. It lacks support for data flow and other dynamic behavior. For example, based on current AOP technology, it is easy to track when a field is accessed but there is no straightforward way to track where the value of the field is passed to and how the value is used in a program.

SONAR does not explicitly support distribution since, currently, no AOP framework provides built-in support for distribution – aspects cross distributed applications/systems. However, by using JMX, aspects deployed in a distributed fashion could ultimately be accessed and managed through JMX Remote API.

## 5. RELATED WORK

### 5.1 Pinpoint

Pinpoint [2,3] is a dynamic analysis methodology that automates problem determination in complex systems by coupling coarse-grained tagging of client requests with data mining techniques. Data mining correlates the believed failures and successes of these client requests as they pass through the system. This combined approach is used to determine which component(s) are most likely to be at fault. Pinpoint has been implemented and used as a framework for root-cause analysis on the J2EE platform that requires no knowledge of the application components.

Pinpoint consists of a communications layer that traces client requests, a failure detector that uses traffic-sniffing and middleware instrumentation, and a data analysis engine. It would be possible to merge Pinpoint and SONAR in such a way that would eliminate the manual instrumentation currently required for Pinpoint's middleware instrumentation.

### 5.2 Magpie

Magpie's [8] goal is to provide synthesis of runtime data into concise models of system performance [8]. In Magpie, online performance modeling is an operating system service. Magpie's modeling service collates detailed traces from multiple machines, extracts request-specific audit trails, and constructs probabilistic models of request behaviour. . It would be possible to adopt some of the strategies used by Magpie that apply to distribution and performance debugging to extend SONAR to further environments.

### 5.3 DTrace

DTrace [4] is a unified tracing toolkit for both system and application levels. DTrace can be used to observe, debug and tune system using the D programming language which is designed specifically for tracing. As a result, it is a comprehensive dynamic tracing framework, though only applicable within the Solaris Operating Environment. DTrace attains many of the goals shared by SONAR, but within this proprietary environment.

### 5.4 PEM/K42

*Vertical Profiling*[3] is an approach to correlating performance and behavior information across the layers of modern systems (hardware, operating system, virtual machine, application server, and application) to identify causes of performance problems. The Performance and Environment Monitoring (PEM) group and K42 group [6, 9] at IBM Research are getting promising results using this among other approaches to develop effective system diagnosis tools. We believe SONAR to be an early prototype of a tool that would fit with this family.

## 6. FUTURE WORK

As mentioned above, both DTrace and PEM/K42 provide unified dynamic tracing across operating system, virtual machine and application. The current SONAR implementation lacks an easy way to instrument the operating system and certain aspects of virtual machine. As for a future improvement, we plan to investigate how to unify SONAR with such tool kits in order to provide an efficient means for tasks requiring flexible and customizable tracing crossing all layers in the software stack, but still focusing more on higher level such as application and middleware. Additionally, we plan to further investigate advances in AOP, especially dynamic AOP. We also hope to eventually integrate a high level language specifically for optimization and navigation into SONAR.

## 7. CONCLUSION

SONAR uses XML, AOP, and JMX in order to achieve a language/framework agnostic, dynamic, manageable, unified framework for system-wide analysis at runtime. We believe the results we have gathered from the prototype to date to be promising in terms of both functionality and performance.

## 8. ACKNOWLEDGMENTS

We would like to thank Jonathan Appavoo for his inspiration and insights.

## 9. REFERENCES

[1] AspectWerkz, http://aspectwerkz.codehaus.org/index.html.

[2] Chen, M., Kiciman, E., Accardi, A., Fox, A., Brewer, E. Using Runtime Paths for Macroanalysis. In HotOS IV (2003).

[3] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, Internet services. Proc. International Conference on Dependable Systems and Networks (IPDS Track), pages 595-604, June 2002.

[4] DTrace, http://www.sun.com/bigadmin/content/dtrace/

[5] JMX, http://java.sun.com/products/JavaManagement.

[6] K42, http://www.research.ibm.com/K42/

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, Aspect-Oriented Programming, European Conference on Object-Oriented Programming (ECOOP), 1997.

[8] Magpie, http://research.microsoft.com/projects/magpie/.

[9] PEM, http://www.research.ibm.com/vee04/Duesterwald.pdf

---

[3] http://www-plan.cs.colorado.edu/~hauswirt/Research/