SONAR: Customizable, Lightweight Tool Support to Prevent Drowning in Diagnostics

Chunjian Robin Liu, Celina Gibbs, and Yvonne Coady
University of Victoria, Canada
{cliu, celinag, ycoady}@cs.uvic.ca

Abstract. Traditional system diagnostic and management techniques rely on static system structure and static instrumentation. We believe that static approaches are simply no longer sustainable in today's complex, distributed and dynamic systems. *SONAR* (Sustainable Optimization and Navigation with Aspects at Runtime) is a fluid and unified framework that allows stakeholders to dynamically explore and modify meaningful entities that are otherwise spread across predefined abstraction boundaries. Through a combination of dynamic Aspect-Oriented Programming (AOP), Extensible Markup Language (XML), and Java Management Extensions (JMX), SONAR can comprehensively coalesce scattered artifacts – enabling iterative and interactive system-wide investigation and control. This paper overviews the motivation, design and implementation of SONAR.

1 Introduction

Runtime behaviour of today's complex systems is increasingly difficult to understand. One of the contributing factors is the increase in dynamism on all fronts – programming languages, frameworks/middleware/virtual machines, operating systems, are all involved in today's adaptive and autonomic systems. Static diagnostic techniques are no longer sustainable in these contexts. For example, Java reflection adds a level of indirection in order to achieve a higher degree of flexibility, but at the same time this makes realizing which object is actually invoked more difficult than a static call. In the context of pervasive systems, which require a high degree of dynamism in terms of device/service states and availability, static based techniques are equally unsuitable for analyzing runtime behaviour.

Another contributing factor to the problem is that of heterogeneity and predefined boundaries. Though layering, componentization, and virtualization provide necessary levers for abstraction, abstraction boundaries and potential for emergent behaviour impair the efficacy of local reasoning. For example, consider root-cause fault-analysis when application level exceptions are absorbed by middleware and hidden from both lower layers and users. Or, similarly, when lower level exceptions are transformed to a different representation for higher levels to digest [2, 3]. Understanding systematic behavior thus requires diagnostic tools that can flow freely across boundaries and provide comprehensive probe data that can be easily collected and correlated.

Finally, looking at this problem from yet another angle, complex system architectures must be designed and documented from the perspective of multiple views for different stakeholders [12]. Traditional perspectives of control flow and data flow must be considered in concert with structural views (classes and deployment) and developed alongside behavioral views (use cases, sequences, and collaboration). When these design views are mapped onto code level artifacts, structural views tend to be direct and explicit, while behavior views are typically more difficult to realize. Furthermore, users with different roles, i.e. designer, developer, administrator and maintainer, will inevitably have different interests, thus requiring different views of a single system at runtime. Views may need to be iteratively refined, as focus changes during the process of analyzing interests [4]. Ideally, infrastructure to support views should be able to be easily removed once users no longer need them, and further incur little to no performance penalty.

SONAR (Sustainable Optimization and Navigation with Aspects at Runtime) is a fluid and unified framework that allows stakeholders to dynamically explore and modify meaningful entities that are otherwise spread across predefined abstraction boundaries. Through a combination of dynamic Aspect-Oriented Programming (AOP), Extensible Markup Language (XML), and Java Management Extensions (JMX), SONAR can comprehensively coalesce scattered artifacts, enabling iterative and interactive system-wide investigation and control. SONAR allows stakeholders to easily shift focus between coarser/finer grained, or even crosscutting entities, and presents system diagnostics in a comprehensive, manageable unit.

1.1 SONAR Requirements

In an effort to provide a sustainable solution to the problems encountered when trying to comprehend behaviour of complex systems, SONAR was designed with three key requirements in mind:

- **Dynamic instrumentation:** Instrumentation code must be able to be inserted and removed while the system is running, and inherently structure the crosscutting nature of the instrumentation for ease of management. Furthermore, once such code is removed, it should have zero impact.

- **Language/framework agnostic definition:** To be able to define entities and data of interest across a spectrum of system elements implemented in a variety of programming languages, instrumentation must be language/framework independent.

- **Semantic representation:** To comprehensively maneuver and manage diagnostics and modifications, data must be aggregated into a semantic representation that corresponds to a stakeholder's interest, and visualized/managed through easy to use standard-compliant tools.

1.2 Related Work

Given the growing need for more holistic diagnostic tools, it is no surprise the SONAR is one of many projects working to address this and related challenges. Within this spectrum, SONAR sits as a lightweight dynamic approach, which could effectively be used in concert with several more heavyweight approaches.

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. Magpie's [8] goal is to provide synthesis of runtime data into concise models of system performance. 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. PEM/K42 [10] uses an approach called *vertical profiling* to correlate performance and behavior information across all layers of a system (hardware, operating system, virtual machine, application server, and application) to identify causes of performance problems. The Performance and Environment Monitoring (PEM) group and K42 Operating System groups [6, 9] at IBM Research are getting promising results using this and a set of other approaches to develop effective system diagnosis tools. Prose [11] is an adaptive middleware platform for dynamic AOP which allows aspects to be woven, unwoven or replaced at runtime.

1.3 Background

In SONAR, dynamic aspects are used to provide runtime instrumentation that supports a crosscutting structure; XML is used as a language/framework agnostic language to fit with multiple AOP frameworks; and finally, JMX provides standard-compliant visualization/management.

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, 13]. Dynamic AOP allows aspects to be introduced/removed to/from a system at runtime. SONAR uses AspectWerkz [1] to provide dynamic AOP to structure system-wide crosscutting concerns for dynamic analysis, and introduce/remove them at runtime.

Extensible Markup Language (XML) was originally designed to improve the functionality of the Web by providing more flexible and adaptable information identification [18]. SONAR uses an XML representation of aspects to achieve an AOP agnostic notation. An interesting technical advantage to this approach is that it allows SONAR to leverage XSL Transformations (XSLT) [19], a language for transforming XML into other documents, and a wide variety of XML-processing tools. SONAR uses XML's ability to encapsulate information in order to pass it between different computing systems that would otherwise be unable to communicate.

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 uses JMX's support for dynamic aspect management and state querying through standard management features.

2 Design and Implementation

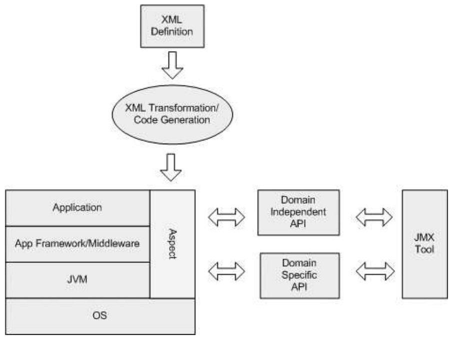


Figure 1. SONAR architecture, showing a high level overview of how an XML definition of an aspect can be introduced to crosscut a system. That is, application, middleware and even the JVM, and subsequently be available for visualization and management through a JMX tool.

The overall architecture of SONAR is organized as shown in Figure 1. Dynamic aspects are generated from XML-based definition files, and can be deployed to applications/frameworks/middleware and virtual machine. They can then be managed through JMX compatible tools. The APIs upon which the JMX tools manage the aspects break down into two distinct categories: (1) domain independent APIs, such as those related to deployment/removal of aspects, and (2) domain specific APIs, such as those related to system navigation. In short, the Figure shows that SONAR's dynamic aspects are produced by XML transformation/code generation, and woven into the target code at runtime. From there, JMX can be used to visualize and manage the system. The following subsections provide more detail on dynamic AOP, XML and JMX respectively.

2.1 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 joinpoint model provides a solid foundation for implementing instrumentation, covering 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 be exposed for instrumentation.

Dynamic AOP further provides a powerful mechanism for runtime aspect manipulation such as runtime deployment/removal. In other words, advice can be dynamically woven into targets and dynamically removed from targets. The current implementation of SONAR uses AspectWerkz[1] to provide dynamic AOP.

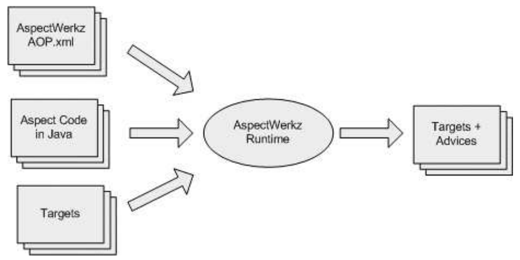


Figure 2. XML configuration files, aspects, and targets are fed into the AspectWerkz runtime weaver to produce targets and advice.

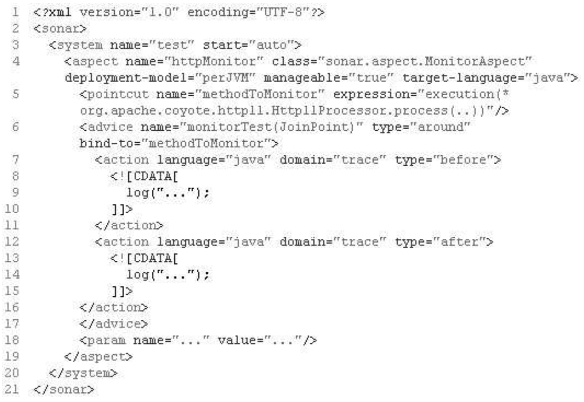
Figure 2 depicts how three key ingredients come together to form the instrumented system: an XML configuration file, aspects and the target system to be diagnosed. The following subsections describe details on SONAR's use of XML and JMX respectively.

2.2 XML Transformation/Code generation

Aspects in SONAR 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 or C++. Figure 3 shows a sample XML definition for an *httpMonitor* aspect. This aspect essentially monitors executions of the *process* method in the *HttpllProcessor* class.

The core content of every aspect specified in SONAR includes: variable/method declaration, pointcut expressions, advice with actions, and parameter definitions. Variable/method declarations and actions are discussed in details in the following subsections. The current schema is mainly based on AspectWerkz's aspect definition schema, which is similar to other existing AOP frameworks. Additional elements of the schema are required are for transformation and code generation.

More specifically, line 3 specifies the target system name, and that the aspect will be started automatically ("auto"). Automatically started aspects are enabled when the target systems are loaded, while manually started aspects have to be explicitly manually enabled (through JMX management tools) at the runtime. Line 4 specifies that there is exactly one of these aspects per JVM. Line 5 identifies a pointcut (*methodToMonitor)* associated with the execution of the *process* method, and lines 6 through 17 define the functionality (around advice, bound to the pointcut in line 5) to be applied. Line 18 defines the parameters to use.



```
 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 showing around advice.

Since the target domain in SONAR is optimization and navigation, variable/method declarations and actions contain code targeting a domain specific API. This defines the boundary between the aspect code and the domain specific target implementation. For example, the log method used in the Figure 3 (lines 9 and 14) 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. In SONAR's current implementation, declarations and actions are written only in Java.

As shown in Figure 4, 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 must be used to compile the code into the language used in the target system. SONAR can accommodate this kind of heterogeneity by supplying additional transformers, one per target language.

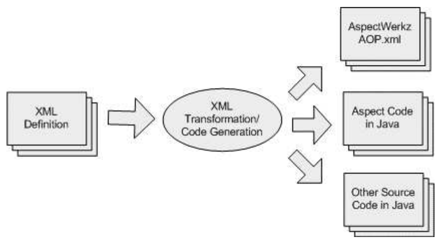
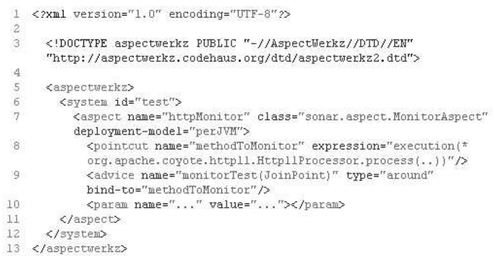

XML Definition
XML Transformation/ Code Generation
AspectWerkz AOP.xml
Aspect Code in Java
Other Source Code in Java

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.

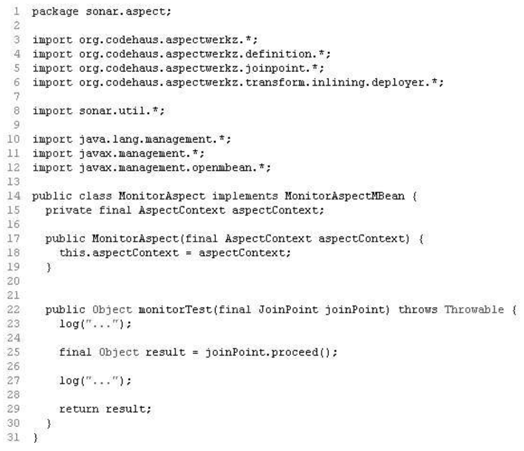


```
<?xml version="1.0" encoding="UTF-8"?>

  <!DOCTYPE aspectwerkz PUBLIC "-//AspectWerkz//DTD//EN"
  "http://aspectwerkz.codehaus.org/dtd/aspectwerkz2.dtd">

  <aspectwerkz>
    <system id="test">
      <aspect name="httpMonitor" class="sonar.aspect.MonitorAspect"
      deployment-model="perJVM">
        <pointcut name="methodToMonitor" expression="execution(*
        org.apache.coyote.http11.Http11Processor.process(..))"/>
        <advice name="monitorTest(JoinPoint)" type="around"
        bind-to="methodToMonitor"/>
        <param name="..." value="..."></param>
    </aspect>
  </system>
</aspectwerkz>
```

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



```
package sonar.aspect;

import org.codehaus.aspectwerkz.*;
import org.codehaus.aspectwerkz.definition.*;
import org.codehaus.aspectwerkz.joinpoint.*;
import org.codehaus.aspectwerkz.transform.inlining.deployer.*;

import sonar.util.*;

import java.lang.management.*;
import javax.management.*;
import javax.management.openmbean.*;

public class MonitorAspect implements MonitorAspectMBean {
  private final AspectContext aspectContext;

  public MonitorAspect(final AspectContext aspectContext) {
    this.aspectContext = aspectContext;
  }


  public Object monitorTest(final JoinPoint joinPoint) throws Throwable {
    log("...");

    final Object result = joinPoint.proceed();

    log("...");

    return result;
  }
}
```

Figure 6. Java source code containing AspectWerkz specific code, as prescribed in Figure 5.

2.3 JMX Management

JMX is used as a means to comprehensively visualize and manage aspects introduced by SONAR. 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 built into Sun's JDK distribution. Figures 7 and 8 show how JConsole can be used to manage aspects.

These Figures specifically show how the simple *MonitorAspect*, which monitors HTTP requests, database access and JSP service, is visualized and managed in SONAR. Figure 7 illustrates how the statistics from three different invocation points collected by MonitorAspect can be visualized as line charts in JConsole. As these data points are updated in the running system, the charts are automatically updated as well, and accessed through the *attributes* tab of the JConsole interface (top left in the Figure).

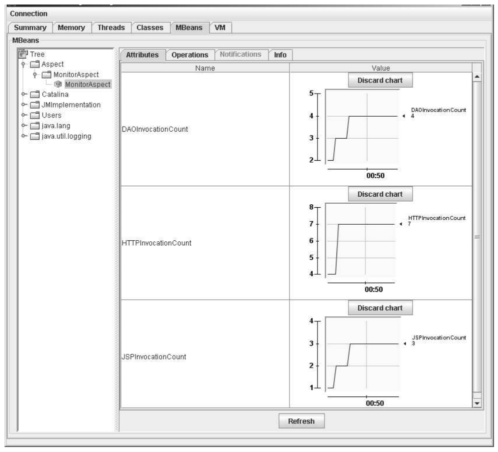


Figure 7. Data from *MonitorAspect* is comprehensively visualized as data values change over time, and can be updated in JConsole.

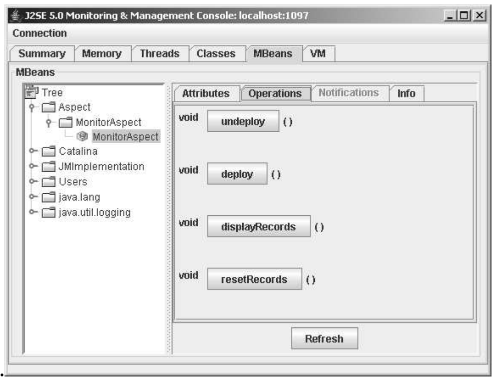


Figure 8. Operations (both generic and domain specific) of *MonitorAspect* are listed and can be invoked manually in JConsole.

Figure 8 shows the operations supported by MonitorAspect. The deploy()/undeploy() buttons are used to manage the aspect at runtime. After being undeployed, advice defined in MonitorAspect is removed from all targets. MonitorAspect still can be accessed by JConsole, and can be redeployed by invoking deploy() from this management interface (*operations* tab, shown at the top of the Figure).

3 Case Study: Tomcat/JBoss/Duke's Bank

The following *TraceAspect* demonstrates SONAR's ability to navigate and optimize a sample system. The example is based on a modified version of the J2EE sample bank application, *Duke's Bank*. Given that the scenario considered is request-centric, a modification to the code to provided a tag for each request. That is, a unique identification (ReqID) is assigned to every request, be it generated from a servlet or JSP access, and sent to the server. The ReqID is retrieved when a request reaches Tomcat's

HTTP adapter, and saved to a thread local variable in order to be accessed by subsequent operations.

In order to demonstrate SONAR's ability to freely navigate across abstraction boundaries in the system, we developed a *TraceAspect*, not enabled (manual) when the system is started. As with all aspects in SONAR however, the TraceAspect itself is registered as a standard MBean (management bean) to the JBoss's JMX server. The stakeholder can thus enable the tracing through domain independent AOP (deploy/undeploy) shown in Figure 8.

The TraceAspect reflects a request-centric view of the system, recording key data points as requests are serviced. As a result, it exposes several key configurable options and operations through JConsole, such as the ability to:

- enable/disable tracing to specified classes and/or methods, including those in JBoss, Tomcat, and the bank application
- apply a filter, consisting of a ReqID pattern, to filter out unwanted data
- configure tracing details (stack trace, timestamps, duration)
- manipulate buffer operations (change the size, clear the buffer)

All the above options and operations are accessible through this aspect's JMX interface. All data is stored at the server side and can be retrieved and viewed through JMX management tool.

Figure 10 visually depicts the information collected by SONAR regarding a stack trace of serving a request to retrieve customer accounts. Each blue bar indicates the processing time (in milliseconds) of a method, the summation of the time spent in processing its method body and subsequent method calls. The top level is the Tomcat's adapter – the entry point of serving a http request. Access to SessionBeans, EntityBeans and JSPs are traced to clearly show the processing time in each layer according to J2EE architecture.

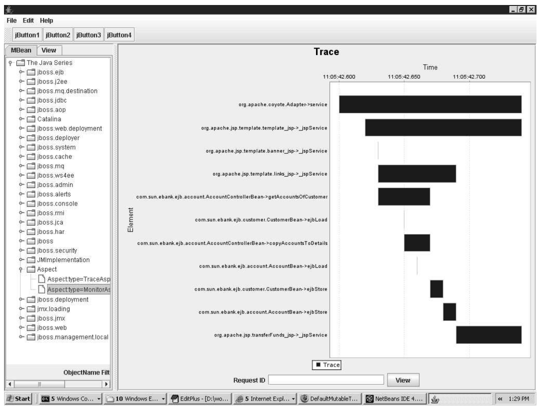


Figure 10. The stack trace of serving a request that retrieves a list of accounts of a customer. The blue bar indicates the processing time (in milliseconds) of each method, the summation of the time spent in processing its method body and subsequent method calls.

4 Conclusions

This paper shows the ways in which SONAR's use of AOP, XML, and JMX as cornerstone technologies to achieve a dynamic, language/framework agnostic, manageable framework are be promising in terms of both functionality and performance. SONAR's code-generator produces low-level code artifacts needed for instrumentation and navigation of higher-level stakeholder views. AOP allows for dynamic weaving of instrumentation aspects, XML supports a language-independent notation for the instrumentation aspects, and JMX provides management and visualization of these aspects.

References

[1] http://aspectwerkz.codehaus.org/

[2] Chen, M., Kiciman, E., Accardi, A., Fox, A., Brewer, E. Using Runtime Paths for Macroanalysis. In the Symposium on Hot Operating Systems, 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] http://java.sun.com/products/JavaManagement/

[6] 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] http://research.microsoft.com/projects/magpie/

[9] http://www.research.ibm.com/CPO/

[10] http://www.research.ibm.com/vee04/Duesterwald.pdf

[11] http://prose.ethz.ch/

[12] Paul Clements; Felix Bachmann; Len Bass; David Garlan; James Ivers; Reed Little; Robert Nord; & Judith Stafford, Documenting Software Architectures: Views and Beyond, ISBN: 0201703726, 2002.

[13] AspectJ, Eclipse Project, IBM, http://eclipse.org/aspectj/

[14] http://jakarta.apache.org/tomcat/

[15] Heisenberg, Werner. *Across the frontiers*; translated from the German by Peter Heath. 1990, http://encyclopedia.laborlawtalk.com/Werner_Heisenberg

[16] Remi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Segura-Devillechaise and Mario Sudholt *An expressive aspect language for system applications with Arachne*, in Proceedings of the 4th international conference on Aspect-oriented software development, ACM Press, Chicago, USA, March 2005.

[17] Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects, Michael Engel and Bernd Freisleben, *Distributed Systems Group, Dept. of Mathematics and Computer Science, University of Marburg*, The International Conference on Aspect-Oriented Software Development, 2005

[18] http://www.w3.org/XML/

[19] XSL Transformations, XSLT, http://www.w3.org/Style/XSL/