SONAR: Safe and Sound Dynamic Analysis

by

Chunjian Robin Liu
B.Sc., University of Victoria, 2004

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Chunjian Robin Liu, 2010
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.
SONAR: Safe and Sound Dynamic Analysis

by

Chunjian Robin Liu
B.Sc., University of Victoria, 2004

Supervisory Committee

Dr. Y. Coady, Supervisor
(Department of Computer Science)

Dr. J. Weber-Jahnke, Departmental Member
(Department of Computer Science)

Dr. H. A. Muller, Departmental Member
(Department of Computer Science)
ABSTRACT

Traditional diagnostic and optimization techniques typically rely on static instrumentation of a small portion of an overall software system. Unfortunately, solely static and localized approaches are simply no longer sustainable in the evolution of today’s complex and dynamic systems. SONAR (Sustainable Optimization and Navigation with Aspects for system-wide Reconciliation)\(^1\) is a fluid and unified framework that enables stakeholders to explore and adapt meaningful entities that are otherwise spread across predefined abstraction boundaries. Through a combination of Aspect-Oriented Programming (AOP), Extensible Markup Language (XML), and management tools such as Java Management Extensions (JMX), SONAR can comprehensively coalesce scattered artifacts—enabling evolution to be more inclusive of system-wide considerations by supporting both iterative and interactive practices. This system-wide approach promotes the application of safe and sound principles in system evolution. In this work, we present SONAR’s model, examples of its concrete manifestation, and an overview of its associated costs and benefits. Case studies demonstrate how SONAR can be used to identify performance bottlenecks accurately and evolve systems by optimizing behaviour effectively, even at runtime.

\(^1\)In seafaring terms, sonar systems ensure safety by using sound waves to detect underwater obstacles.
# Contents

Supervisory Committee \hfill ii

Abstract \hfill iii

Table of Contents \hfill iv

List of Tables \hfill vii

List of Figures \hfill viii

Listings \hfill x

Acknowledgements \hfill xi

Dedication \hfill xii

1 Introduction \hfill 1
    1.1 Proposed Approach: SONAR \hfill 2
    1.2 Related Work \hfill 4
        1.2.1 Pinpoint \hfill 4
        1.2.2 Magpie \hfill 6
        1.2.3 DTrace \hfill 8
        1.2.4 JFluid \hfill 10
        1.2.5 PEM/K42 \hfill 11
        1.2.6 PROSE \hfill 12
        1.2.7 Others \hfill 13
        1.2.8 Summary \hfill 14
    1.3 Thesis Outline \hfill 15

2 Background: Instrumentation and Management Tools \hfill 16
2.1 Modularity and Instrumentation ........................................ 16
  2.1.1 Crosscutting Concerns ........................................... 16
  2.1.2 Object-Oriented Programming ................................... 19
  2.1.3 Domain-specific Solutions ................................. 19
  2.1.4 AOP ............................................................. 20
2.2 Language Agnostic Instrumentation ................................. 21
  2.2.1 XML Namespaces ................................................ 21
  2.2.2 XML Schema .................................................... 21
  2.2.3 XSLT ............................................................. 22
2.3 Management Tools .................................................... 23
  2.3.1 Instrumentation Layer .......................................... 25
  2.3.2 Agent Layer ..................................................... 25
  2.3.3 Distributed Layer .............................................. 26
  2.3.4 Manager Layer .................................................. 27
2.4 Summary ............................................................. 27

3 Design and Implementation ............................................ 28
  3.1 Dynamic AOP Integration .......................................... 30
  3.2 XML Transformation/Code generation ........................... 31
  3.3 JMX Management .................................................... 35
  3.4 Summary ............................................................. 38

4 Supporting Infrastructure ............................................... 39
  4.1 Flows ............................................................... 40
    4.1.1 Data Structure ................................................ 41
    4.1.2 Flow Management .............................................. 44
    4.1.3 Flow Propagation ............................................. 45
    4.1.4 Flow Branching ............................................... 48
  4.2 Probes ............................................................. 51
    4.2.1 Data Structure ................................................ 51
    4.2.2 Probe Management and Process .............................. 54
    4.2.3 Probes meet Flows .......................................... 56
  4.3 Logs ............................................................... 57
    4.3.1 Why logs? ..................................................... 57
    4.3.2 Existing Logging Frameworks ................................. 58
5 Case Studies: Optimization and Navigation with SONAR

5.1 Operating System Optimization

5.2 Virtual Machine Navigation

5.3 Application Level Navigation and Optimization

5.3.1 Navigation of a Simple Banking Application

5.3.2 Cross-Platform Optimization

5.4 Analysis: Costs and Benefits

5.4.1 Costs: Performance and Memory Utilization

5.4.2 Benefits: Programming Model and Unified Framework

6 Future Work and Conclusion

6.1 Contribution

6.2 Future Work

Bibliography
List of Tables

Table 1.1 Summary of related work .................................................. 14

Table 5.1 Impact on target class size. Both classes have a single around
advice woven to a single method, respectively, using AspectWerkz
2.0. ................................................................. 90
List of Figures

Figure 1.1 Pinpoint Framework[12] ........................................ 5
Figure 1.2 Magpie Architecture [9] ..................................... 7
Figure 1.3 DTrace Architecture and Components [47] .......... 9
Figure 1.4 PEM Architecture [21] ................................. 11

Figure 2.1 Crosscutting concerns in a large component [10] .... 18
Figure 2.2 Transformation using XSLT. ................................ 22
Figure 2.3 JMX Architecture [28] .................................... 24

Figure 3.1 SONAR architecture, showing a high-level overview of an XML definition of an aspect. The figure outlines how an aspect can be introduced to crosscut a system and interoperate with associated tool support. ........................................ 29
Figure 3.2 XML configuration files, aspects, and targets are fed to the AOP weaver to produce targets and advice. ............. 30
Figure 3.3 XML definition files are transformed into configuration files and source code in a target language using XSLT and a domain-specific compiler/code generator. .................. 33
Figure 3.4 Data from the MonitorAspect is visualized as data values change over time, and can be updated in JConsole. .... 36
Figure 3.5 Operations (both generic and domain specific) of the MonitorAspect are listed and can be invoked manually in JConsole. 37

Figure 4.1 Flow and related classes ................................. 42
Figure 4.2 Flow setting bits ......................................... 43
Figure 4.3 Flow states and operations .............................. 45
Figure 4.4 Flow in string form ....................................... 46
Figure 4.5 Flow stamp and branching .............................. 49
Figure 4.6 Asynchronous branch ................................. 50
Figure 4.7  Synchronous branch. .......................................... 50
Figure 4.8  Probe and related classes .................................... 52
Figure 4.9  Probe and flow integration ................................. 55
Figure 4.10 Flow-aware logger and factory ............................ 62
Figure 4.11 Logger creation with SONAR ............................... 63
Figure 4.12 Flow-aware logging ......................................... 65

Figure 5.1  Rudimentary data aggregation with vmstat ............. 72
Figure 5.2  Rudimentary data aggregation with mpstat ............. 72
Figure 5.3  This screenshot shows GCspy visualizing Sun’s Java HotSpot virtual machine running the SPECjvm98_213_javac benchmark. 73
Figure 5.4  The stack trace of serving a request that retrieves a list of customer accounts and their corresponding balances. ........ 83
Figure 5.5  Impact on startup time. Sampled by running JBoss Application Server 4.0.1 SP1 with embedded Tomcat 5 and AspectWerkz 2.0. 91
Figure 5.6  Impact on memory footprint. Sampled under the same setting as in Figure 5.5. ................................. 92
Listings

3.1 Sample XML definition file showing an around advice .......................... 31
3.2 AspectWerkz’ aop.xml, generated by transforming the definition file in Listing 3.1 ................................................................. 34
3.3 Java source code containing AspectWerkz-specific code, as prescribed in Listing 3.2 ................................................................. 34
4.1 Flows usage example ........................................................................ 56
4.2 Probes usage example ...................................................................... 57
5.1 XML specification for a prefetching aspect for the AspectC transformer 68
5.2 Prefetching aspect generated in AspectC ................................. 70
5.3 XML specification for GCSpy aspect for AspectJ transformer ...... 73
5.4 GCSpy aspect generated in AspectJ ............................................. 74
5.5 Request intercepting aspect generated in AspectWerkz ............. 78
5.6 RequestMatcher ........................................................................... 79
5.7 Flow-aware profiling aspect generated in AspectWerkz ............ 80
5.8 Log from serving a flow-attached request ..................................... 84
5.9 XML specification of cache aspect for Spring.NET transformer ... 87
5.10 Cache aspect generated in Spring.NET ........................................ 88
5.11 Configuration XML generated in Spring.NET ............................ 88
ACKNOWLEDGEMENTS

First and foremost, I am heartily thankful to my supervisor, Dr. Yvonne Coady, for her unparalleled guidance, support and encouragement, through all these years. Her knowledge and insight have been invaluable to my research. Without her inspiration and confidence in me, I would not be able to make to the end. I would also like to thank Onon Chak and Hitomi Nakahara for their support and understanding, especially during my low times. Lastly, I offer my regards and blessings to all of those who have helped, encouraged and inspired me in any respect during my study.
DEDICATION

I would like to dedicate this thesis in honor of my parents for their unconditional love, constant inspiration, unreserved support, and faith in me.
Chapter 1

Introduction

Today’s complex systems’ behaviours are increasingly difficult to understand and anticipate. One of the contributing factors is the increase in subtle interactions on all fronts—frameworks, middleware, virtual machines, and operating systems. Purely static techniques for evolution are often no longer sustainable in these contexts. Heterogeneity and predefined abstraction boundaries are obstacles to system evolution. Though layering, componentization, and virtualization provide necessary levers for abstraction, behaviour that emerges along execution paths crossing these boundaries ultimately extends beyond simple localized reasoning. For example, consider the ordeal of trying to find the root-cause of faults in a system. It is not uncommon for meaningful application-level exceptions to be absorbed by middleware in a distributed system, and thus hidden from view [11]. Or, similarly, lower-level exceptions can sometimes be transformed to a different representation for higher levels to handle, making it more difficult to diagnose the root-cause of failure [12]. These scenarios highlight the need to reconcile issues such as fault analysis in terms of a system as a whole, as it is inadequate to try to diagnose problems when limited to one layer or component. We believe the same principle holds in the context of system evolution. That is, it is often difficult to evolve a system in a safe and sound manner without considering the system as a whole.

Understanding and evolving system behaviour thus requires approaches that can flow freely across boundaries and provide comprehensive analysis that can be easily collected, correlated, and ultimately used to adapt applications to new circumstances, sometimes dynamically, as they evolve. Looking at this problem from another angle, complex system architectures must be viewed from multiple perspectives for multiple
stakeholders [13]. Furthermore, in order to aggregate data effectively, views may need to be refined iteratively as focus changes during the process of analyzing interests [19]. Ideally, infrastructure to support views should be easily removed once users no longer need it, and hence incur little to no performance penalty. Recent technologies such as those employed by JFluid [44] go a long way to demonstrating that dynamic bytecode instrumentation can be both customized and efficient.

We argue that, for a large class of optimization strategies related to unanticipated external environment conditions, optimizations are becoming an increasingly substantial obstacle to effective evolution. Mixing optimization logic with application logic requires non-local information and makes both of them more difficult to understand, maintain, and evolve, due to idiosyncratic dependencies on external factors. Further, optimization code is context dependent and highly sensitive to dynamic factors such as server load, network traffic, and even order of operation completion. These factors make it particularly inefficient to encode certain kinds of optimizations in the absence of a-priori knowledge about execution contexts in terms of system-wide optimizations. We thus believe tool support to enable system-wide diagnosis and optimization must allow developers to apply and reconcile system monitoring and optimization techniques globally—across operating systems, virtual machines and applications. Further, at the application level, context-specific optimizations applied at runtime can supply much needed support for highly sensitive dynamic factors.

1.1 Proposed Approach: SONAR

SONAR (Sustainable Optimization and Navigation with Aspects for system-wide Reconciliation) is a fluid and unified framework that allows stakeholders to dynamically explore and adapt meaningful entities that are otherwise spread across predefined abstraction boundaries. This allows for a safe and sound approach to system evolution—safe because it is informed, and sound because it is principled. SONAR is fluid in that it can leverage aspects to flow across boundaries in the system including the operating system, virtual machines, and applications, and it is unified in that it provides a language-agnostic, holistic approach to diagnosis and optimization.

Through a combination of Aspect-Oriented Programming (AOP), Extensible Markup Language (XML), and management tools ranging from low-level system calls to high-
level features such as Java Management Extensions (JMX), SONAR can comprehensively coalesce scattered artifacts. This enables iterative and interactive system-wide investigation and subsequently safe evolution. SONAR allows a view of the system to easily shift focus between coarser/finer-grained entities along principled points of execution paths that cross abstraction boundaries. At the application level, SONAR’s model of deployment includes dynamic application of aspects to address the increasing need for runtime optimizations that can be customized to execution environments. As the deployment of such optimizations presents a new set of management challenges, SONAR further offers centralized support for a dynamic aspect repository to help prevent a disjointed view of the system as it is being altered at runtime.

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

- **Principled and system-wide instrumentation**: Instrumentation code must be introduced at principled points in the execution of the system. In order to be system-wide, these points must include all layers of the software stack, such as the operating system, the virtual machine, and the application. Furthermore, in the case of applications, dynamic instrumentation must be supported in a way that lends itself to centralized management. Finally, in the case of low-level infrastructure, instrumentation must have zero impact if disabled or removed. That is, there should be no residual scaffolding left behind.

- **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 optimizations, data must be available for aggregation into a semantic representation that corresponds to a stakeholder’s interest, and visualized/managed through easy to use standard-compliant tools when available. Alternatively, when such tools are not available, the introduction of customized tools or low-level interfaces to system diagnostic techniques must be supported in a way that can be extended to support further comprehensibility or filtering.
Given these requirements, we believe the SONAR model enforces system-wide evolution that is both better informed and principled than an ad hoc strategy. It is safe in that evolutionary changes can be accompanied by a more informed system-wide perspective, and sound because it is based upon principled instrumentation strategies that can be replicated over the lifetime of the system. The main contributions of SONAR are:

- A model for a unified framework: with this model, instrumentation can be defined in a structured and uniform way which facilitates understanding, management, reuse and extension.

- Infrastructure to support the model in distributed systems: allows artifacts such as logs can be selectively generated, collected and later correlated for dynamic analysis purposes.

- Prototype elements of model assessed in a feasibility study: demonstrate the possibility and feasibility of SONAR’s model and infrastructure in certain environments.

1.2 Related Work

Given the growing need for more holistic, system-wide, diagnostic tools, it is no surprise that 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.

1.2.1 Pinpoint

Pinpoint [11, 12] is a dynamic analysis methodology that automates problem determination in dynamic, distributed systems by coupling coarse-grained tagging of client requests with data mining techniques. As client requests pass through a system, believed failure or success and component(s) involved in serving these requests are logged. Data mining uses data clustering and statistical techniques to correlate failures of client requests to components. This combined approach is used to determine which component(s) are most likely to be at fault, and has been applied to the problem of root-cause analysis on the J2EE platform with impressive results. As shown in Figure 1.1, Pinpoint consists of three parts:
Figure 1.1: Pinpoint Framework[12]
• Client request tracing: The communication layer traces each individual client request through the system and tracks which component(s) are involved in serving the request. This is provided by instrumenting the middleware and communication layer between components. Each request is uniquely identified by a global request identifier which is propagated through the system using modified middleware (i.e., J2EE server).

• Failure detection: The failure detectors attempt to detect whether client requests fail. They use middleware instrumentation and traffic-sniffing for detecting both internal (e.g., assertions/exceptions in application components) and external failures (e.g., network outage). Failure detectors log the detected status along with the request identifier.

• Data Analysis: The data analysis engine runs a data clustering algorithm on the client request traces and failure/success logs to correlate failures of client requests to components. The components whose occurrences are most correlated with failures are most likely the root cause.

In terms of tracing client requests, Pinpoint and SONAR are very similar. But SONAR provides a more sophisticated mechanism to identify each sub-processing path uniquely. Moreover, Pinpoint provides data mining where SONAR simply offers an iterative and interactive interface for human control. In terms of failure detection, Pinpoint uses traffic sniffing and manually provided middleware instrumentation. SONAR again relies on human interaction to navigate to points of interest, and dynamically deploys/removes aspects for instrumentation. We believe a future merger of these two approaches could provide the best spectrum of support for complex system diagnosis.

1.2.2 Magpie

Magpie [40], developed by Microsoft Research, is a performance analysis tool for distributed systems. It is designed to provide synthesis of runtime data into concise models of system performance. It features a bottom up, per-request approach which is based on the following observations:

• Existing techniques (i.e., performance counters, program profiling) are insufficient to diagnose problems like the system as a whole performs well, yet indi-
individual users see poor performance.

- Accurate diagnosis of the above problem depends on a detailed audit trail of each request and a model of normal request behaviour.

Magpie tracks individual requests (e.g., HTTP request, database query) end to end with relevant data (e.g., control flow and resource consumption). Based on aggregated request data, it builds a probabilistic workload model which can be used for performance debugging, capacity planning, tuning and anomaly detection.

In Magpie, online performance modeling is an operating system service. Magpie’s modeling service collates detailed traces from multiple machines, extracts request-specific information, and constructs probabilistic models of request behaviour. As illustrated in Figure 1.2, Magpie consists of three main components:

- Instrumentation: The instrumentation logs system activities such as kernel level events (e.g., context switch and I/O operation events) and user level events (e.g., enter into and exit from selected procedures in application and middleware). Magpie’s black-box instrumentation has low-overhead and requires no source code change to the target system.

- Request extraction: In Magpie, a request is described by its path taken through the system components together with its resource consumption (i.e., CPU, disk
accesses and network bandwidth usage). Request parser is used for extracting individual request activity from interleaved event traces. The request parsing algorithm relies on an event schema which describes the semantics of events for the particular application [29]. This schema-based mechanism enables the reuse of the same instrumentation and parser to extract different types of requests. Moreover, Magpie uses temporal joins to combine related events into requests instead of using global request identifier throughout the system as seen in Pinpoint.

- Modelling: This process constructs various models, such as behavioural clusters and probabilistic model of request behaviour, based on the extracted request activity data [39]. These models are made available for online programmatic query and can be feed into performance debugging and prediction tools. For example, comparing observed behaviour against the model allows identification of anomalous requests and malfunctioning system components [9].

It would be possible to adopt some of the strategies used by Magpie for distribution and performance debugging within SONAR with the intention of applying it to further environments, in particular operating system services. As SONAR uses language-agnostic definitions for instrumentation points, it could be used to deploy dynamic aspects for C [18, 22] into operating system services.

### 1.2.3 DTrace

DTrace [19], developed by Sun Microsystems, is a unified dynamic tracing toolkit for both system and application levels. It was initially developed for Solaris and later ported to FreeBSD and Mac OS X. The integration of system-level and application-level tracing in DTrace gives operational insights which allow debugging, troubleshooting, and tuning both applications and operation systems. DTrace is carefully designed so that it is safe to use in production environments.

As shown in Figure 1.3, the following are the key concepts in DTrace:

- Probe: A probe is an instrumentation point which can be traced by DTrace. Probes are made available to DTrace by providers.
Figure 1.3: DTrace Architecture and Components [47]
• Provider: A provider represents an instrumentation methodology for a particular area of the system. It makes probes available to DTrace. When a probe is to be enabled, its provider is notified; when an enabled probe is hit, its provider transfers control to DTrace. DTrace provides a number of built-in providers such as the lock stat provider which can dynamically instrument the kernel synchronization primitives. Applications can supply their own providers in order to allow themselves to be traced by DTrace. For instance, Java SE 6 HotSpot JVM introduces providers with probes for monitoring JVM internal state and activities as well as Java applications [1].

• Consumer: A consumer is a process that interacts with DTrace. The command line tool dtrace is a DTrace consumer that acts as a generic front-end to DTrace. Concurrent consumers are supported by DTrace.

• The D programming language: DTrace introduces the D programming language which is similar to C or Awk and defines variables (e.g., pid which is the current process ID) and functions (e.g., aggregation functions) specific to tracing. Tracing programs (scripts) written in D are dynamically interpreted by DTrace when corresponding probe(s) fire. This approach allows arbitrary predicates and actions to be taken when a probe fires.

Some other features of DTrace include: powerful data management primitives (i.e., data aggregation and speculative tracing facility) which eliminate the need for most postprocessing; predicates which allows actions to be taken only if certain conditions are met.

DTrace attains many of the goals shared by SONAR, to monitor, debug and tune systems and runtime from multiple perspectives, but within a proprietary environment. Furthermore, it has no built-in support for tracing in a distributed setting which SONAR aims to provide.

1.2.4 JFluid

JFluid [44] (also known as NetBeans Profiler) is a full-featured profiler built into the NetBeans IDE [43]. It provides profiling functions, such as monitoring CPU, memory and threads, to aid performance-related diagnostics.
JFluid uses highly efficient dynamic bytecode instrumentation, making it possible to use the tool on the fly. A mechanism in the JVM called HotSwap [17] allows users to dynamically turn profiling on/off and to profile just a selected subset of the code. The subset and target of profiling (CPU, memory, etc.) can be changed at runtime. Dynamic bytecode instrumentation is guaranteed not to alter program semantics, as it only impacts well-defined events, such as method entry/exit, and object allocation. JFluid demonstrates the ability to provide highly efficient bytecode instrumentation, which bodes well for costs associated with dynamic AOP.

This analysis of JFluid technology indicates that the implementation of navigation in SONAR could be low cost and localizable, though we expect that the ability to adapt or optimize will come at a cost.

1.2.5 PEM/K42

Performance and Environment Monitoring (PEM) [21] is a research project by IBM Research. It uses an approach called vertical profiling [26] to correlate performance and behaviour information across various layers of a system (i.e., hardware, operating system, virtual machine, application server, and application) to identify, characterize, diagnose, and alleviate performance problems. As shown in Figure 1.4, PEM consists of the following parts:

- Event specification repository: This is a repository of XML-based event specifications which define semantics and attributes of the events in the system.
• PEM toolset: This is a toolset which takes event specifications as input and generate language-specific interfaces and stubs. Currently, C, C++, Fortran, and Java are supported.

• PEMAPI: PEMAPI is an interface between event provider and event consumer. It defines a set of performance-monitoring abstractions: event, event attribute, event set and context. PEMAPI allows event provider and event consumer to be implemented independently.

• Visualization client: This is a vertical performance visualization client that helps users to understand the correlation of events across layers [21].

The PEM and K42 Operating System groups [31] at IBM Research are getting promising results using this and a set of other approaches to develop effective system diagnosis and tuning tools. PEM is implemented on K42, an open source research operating system, and it leverages K42’s efficient tracing facility to gather event information from all system layers [31]. The vertically integrated performance information provided by PEM enables Continuous Program Optimization (CPO) [16] to understand the interaction among various system layers and therefore enables optimizations based on these interactions [15].

Though SONAR in comparison is a much more lightweight approach, we believe it could be an early prototype of a tool that would fit with this family. In particular, the use of JMX in SONAR was inspired by the comprehensive interfaces used by PEM to provide visualization and management of system diagnostics.

1.2.6 PROSE

PROgrammable extenSions of sErvices (PROSE) [45] is an adaptive middleware platform for dynamic AOP which allows aspects to be woven, unwoven or replaced at runtime. It allows systematic modifications to be applied to Java-based applications in a controlled manner at runtime [46]. PROSE features an efficient mechanism to implement dynamic AOP using run-time code replacement. Currently, PROSE supports two JVMs: HotSpot JVM and Jikes RVM. PROSE’s run-time code replacement is implemented by using the HotSwap mechanism in HotSpot JVM and just-in-time compilation (JIT) in Jikes RVM.
PROSE aspects are represented as plain Java objects and can be sent to computers on a network. An interesting feature of PROSE is that it allows aspects to be inserted into or withdrawn from distributed applications transactionally. Once an aspect has been inserted into a JVM it will execute advice as expected until it is removed. PROSE provides a set of join point types (e.g., method entry and exit) as well as a mechanism to let users add new join point types.

PROSE provides a set of middleware tools, named PROSE WORKBENCH, to allow runtime monitoring and visualization of remote aspects, and aspect insertion/withdrawal [46]. Additionally, it provides a creation wizard in its development tools for creating dynamic aspects.

Though dynamic AOP is something common to both PROSE and SONAR, SONAR simply leverages it for dynamic instrumentation at system and/or application levels, whereas PROSE goes much deeper to leverage it for pervasive and coordinated access to the system’s resources. In the long term, these lightweight/heavyweight approaches will converge, allowing high-level management of core system components supplied by aspects.

1.2.7 Others

In terms of convergence, we also see a great number of other approaches to AOP that we would like to incorporate as an option within the SONAR model. CaesarJ [4] is a Java based programming language where components are collaborations of classes. CaesarJ is used to modularize crosscutting features or non-functional concerns better by providing explicit support to implement, abstract and integrate such components. More recently, AWED (Aspects With Explicit Distribution) [41] has proposed an approach for the implementation of crosscutting functionalities in distributed applications with several distinct features specific to a distributed programming domain. These features include remote pointcuts, distributed advice, and distributed aspects, and each support natural notions of state sharing and aspect instance deployment among a group of hosts.

In terms of dynamic AOP and profiling, other important related work includes TOSKANA [22] and Glassbox [25]. TOSKANA provides a toolkit for deploying dynamic aspects into an operating system kernel. It provides before, after and around
advice for kernel functions and supports the specification of pointcuts and the implementation of aspects as dynamically exchangeable kernel modules. Glassbox uses aspect libraries for profiling and troubleshooting Java applications, automatically diagnosing common problems. These services allow developers to spend less time dealing with logging and debugging, and provide an efficient means of performance tuning.

The language-agnostic goal of SONAR is most closely related to Lafferty’s work [37] which supports the AspectJ notion of AOP which is also consistent with the component model of the .NET framework. In this work, aspect-based properties are implemented as Common Language Infrastructure (CLI) components with XML-based crosscutting specifications, and load-time weaving is used. Though the aspect-component bindings are written in terms of attribute types, additional support for custom crosscutting can be specified in terms of CLI metadata, enabling further language-independence. We envision this to be a complementary approach to SONAR’s current approach to transformations, and seek to include this as a further option for transformation in future incarnations of SONAR.

1.2.8 Summary

The related works discussed above are designed to solve various problems. Table 1.1 is the summary of their intended usage and features.

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Scope</th>
<th>Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pinpoint</td>
<td>root cause analysis</td>
<td>distributed system</td>
<td>statically instrumented middleware and communication layers</td>
</tr>
<tr>
<td>Magpie</td>
<td>performance analysis</td>
<td>distributed system</td>
<td>static (schema-based)</td>
</tr>
<tr>
<td>DTrace</td>
<td>dynamic tracing</td>
<td>machine</td>
<td>dynamically selected from pre-defined instrumentation points</td>
</tr>
<tr>
<td>JFluid</td>
<td>profiling</td>
<td>JVM</td>
<td>dynamic using byte code instrumentation</td>
</tr>
<tr>
<td>PEM/K42</td>
<td>performance analysis</td>
<td>distributed system</td>
<td>statically generated interfaces and stubs</td>
</tr>
<tr>
<td>PROSE</td>
<td>adaptive middleware platform</td>
<td>JVM</td>
<td>dynamic using byte code instrumentation</td>
</tr>
</tbody>
</table>
1.3 Thesis Outline

The thesis proceeds as follows. Chapter 2 covers the background of the instrumentation and management tools used by SONAR. Chapter 3 covers SONAR’s high level design and implementation. Chapter 4 provides details on some of the supporting infrastructure used by SONAR. Chapter 5 provides case studies and evaluation of the SONAR model in general, along with a more detailed performance and memory footprint evaluation for dynamic aspects in particular. Chapter 6 concludes with a discussion of future work.¹

¹The core ideas of this work has been published in [38].
Chapter 2

Background: Instrumentation and Management Tools

This chapter briefly introduces the three key technologies used by SONAR to meet the established requirements outlined in Section 1.1.

Both static and dynamic aspects are supported as a means to provide instrumentation that supports a crosscutting structure; XML is used as a language/framework-agnostic language to fit with multiple AOP frameworks; and finally, management tools such as JMX provide standard-compliant visualization and management. The following subsections consider the tension between modularity and instrumentation from the perspective of crosscutting concerns, object-oriented programming, domain specific solutions and aspect-oriented programming, respectively.

2.1 Modularity and Instrumentation

The following subsections consider the tension between modularity and instrumentation from the perspective of crosscutting concerns, object-oriented programming, domain specific solutions and aspect-oriented programming, respectively.

2.1.1 Crosscutting Concerns

A typical software system may consist of several types of concerns such as business logic, performance, persistence, logging and debugging, authentication, security,
multithread safety, error checking, monitoring and management, and development-process concerns (e.g., comprehensibility, maintainability) [35]. Crosscutting concerns are aspects of software systems which affect (crosscut) other concerns. For example, in a bank application, a requirement of logging every bank transaction is a crosscutting concern which must be addressed in all the modules dealing with bank transaction, e.g., deposit and withdrawal modules. Some common crosscutting concerns are security, optimization, distribution, logging, exception handling, transaction management, monitoring and management. Crosscutting concerns usually root from functional/non-functional requirements which are orthogonal to other requirements. They are hard to be cleanly decomposed from the rest of the system in requirements, design and/or implementation [34]. Consequently, they often present in more than one module, and cannot be better modularized through traditional means [34, 6].

The lack of support for modularizing crosscutting concerns results in code scattering or tangling. Code scattering is that multiple modules (e.g., code block, method, class or component) contain similar code addressing the same concern (e.g., logging and exception handling). Scattered code is not localized (modularized). Therefore, it results in tight coupling and low cohesion between those modules, low reusability of shared code. Code tangling is that a single module contains code addressing multiple concerns. For instance, in a code block dealing with bank deposit, programmers usually have to also write code dealing with logging, exception handling, synchronization, transaction and resource management (e.g., begin/commit/rollback transaction, database connection acquisition and release) [35]. Code tangling violates the separation of concerns principle. It makes code hard to write, understand, maintain and reuse. Code scattering and tangling tend to appear together because they describe different facets of the same problem: bad modularity [33]. M. Bruntink et al.’s study shows that code implementing crosscutting concerns tend to involve a great deal of duplication. As illustrated in Figure 2.1, the crosscutting concerns they considered, i.e., tracing (green), pre- and post-condition checking (yellow), memory-error handling (blue), and general-error handling (red), comprise roughly 31 percent of the code in a large component (19,000 lines of code) [10].
Figure 2.1: Crosscutting concerns in a large component [10]
2.1.2 Object-Oriented Programming

Object-Oriented programming (OOP) is the most common programming methodology employed today to develop software systems. It provides the ability to modularize software systems in terms of collaborating objects, as opposed to instructions/procedures. The fundamental concept in OOP is the notion of a class which provides modularity and encapsulation. A class defines the characteristics and behaviours of an object. Along with other concepts, such as inheritance, polymorphism, composition and so on, class allows us to decompose software systems into single-dimensional hierarchical class structures. The strength of OOP is to model common characteristics and behaviors of objects and relationships between objects [35]. However, OOP does not provide adequate support for modularizing crosscutting concerns. More precisely, single, hierarchical decompositions provided by OOP cannot modularize all the concerns in a software system, especially the crosscutting ones [34].

2.1.3 Domain-specific Solutions

In order to relieve application developers from handcrafting some common crosscutting concerns (e.g., object pooling, transaction management and logging), application frameworks abstract them into framework services. These services can be invoked explicitly by application code through service API or implicitly by the framework.

The main limitation of this approach is that it is domain-specific. Each framework only modularizes the crosscutting concerns commonly seen in its own domain. As a result, for application-specific crosscutting concerns which are not addressed by the framework, application developers have to build their own solutions. For instance, the Enterprise JavaBeans (EJB) framework addresses some common crosscutting concerns in enterprise applications, e.g., security, performance, and container-managed persistence. However, it does not provide a solution to modularize the logging concern discussed in Section 2.1.1, which is specific to the example bank application. In short, the domain-specific approach does not provide a general purpose solution for addressing crosscutting concerns.

Furthermore, domain-specific solutions vastly differ from each other in terms of programming model, service API, features, limitations and pitfalls. Application developers have to learn all the above for each solution in order to take advantage of
framework-provided services.

2.1.4 AOP

Aspect-Oriented Programming (AOP) is a new programming methodology aiming at facilitating separation and modularization of crosscutting concerns. AOP complements OOP by introducing a new type of modularity which allows a crosscutting concern to be localized into a single unit, called an aspect. In other words, it provides the ability to divide software systems into multiple, crosscutting decompositions in addition to single, hierarchical decompositions provided by OOP. Each concern can be implemented separately and then integrated with others with absolutely no or minimal coupling [35]. The central concept in AOP is the join point model. The join point model defines join points, pointcuts and advice:

- Join points are well-defined points in the flow of program execution, e.g., method execution, method call.

- Pointcuts are predicates which matches some join points. Just like predicates, pointcuts can be composed using operators (e.g. AND, OR and NOTE) [32].

- Advice provide a means of affecting semantics at join points [33].

The join point model enables coordinating effect of programs from different decompositions [33]. Dynamic AOP allows aspects to be introduced to and removed from a system at runtime.

Since the implementations of concerns are localized (modularized) instead of being scattered, it is much easier to understand, maintain and reuse the code. Simply looking at an aspect, a developer can see both the internal structure of a crosscutting concern and its interaction with the rest of the system during execution.

Nowadays, AOP is used in real projects for enhancing middleware platforms, monitoring and improving performance, adding security to existing applications, and implementing Enterprise Application Integration (EAI) [36]. The current SONAR implementation provides support for AspectC [5], AspectJ [6], AspectWerkz [8] and Spring.NET AOP [51] as means of providing both static and dynamic AOP to structure system-wide crosscutting concerns for analysis and optimization.
2.2 Language Agnostic Instrumentation

Extensible Markup Language (XML), developed by an XML Working Group, is a simplified subset of Standard Generalized Markup Language (SGML). Despite its name, XML itself is not a markup language. Instead, it is a general-purpose metalanguage for creating custom markup languages. XML is platform and language independent. Its primary purpose is to facilitate data sharing across different information systems, particularly systems connected via the Internet[53]. XML is widely adopted as a standard language to structure and describe data. Its flexible tagged structure can be used to store complex data. XML includes specification for namespaces, schema and XSLT, as described in the subsections that follow.

2.2.1 XML Namespaces

As a metalanguage, XML itself does not specify any vocabulary for naming elements and attributes in XML documents. Any company or individual can create their own vocabulary as long as the terms in which conform to XML’s syntactical rules, e.g., no whitespace in element and attribute names. However, this introduces the risk of potential name conflicts. To avoid name conflicts, XML Namespaces is introduced as a means to uniquely name elements and attributes in XML documents. By using namespaces, elements and attributes of different XML document types can be differentiated from each other and therefore can be combined together into other documents.

2.2.2 XML Schema

There are two levels of correctness of an XML document:

- Well-formedness: A data object is an XML document if it is well-formed, as conforming to all the syntactical rules outlined in XML specification.

- Validity: An XML document is valid (or schema-valid) if it conforms to a schema defined in XML Schema or Document Type Definitions (DTD).

XML Schema provide a means to define and describe the structure, content, and semantics of XML documents. Unlike DTD, XML Schema is namespace-aware and XML-based. An XML Schema consists of type definitions and element declarations.
2.2.3 XSLT

XSL Transformations (XSLT) is an XML-based language for transforming XML documents into arbitrary text-based formats (which may or may not be XML). Figure 2.2 illustrates the transformation process using XSLT. An XSLT stylesheet is an XML document processed by an XSLT processor to perform a transformation on other XML documents. An XSLT stylesheet contains one or more template rules for transformation. A template rule contains a pattern which is matched against nodes in the source tree and a template which can be instantiated to form part of the result tree. A template contains literal result elements and XSLT instruction elements which are used to alter the processing of the template. XPath expressions are used extensively in XSLT to select elements for processing, for conditional processing and for generating text [54]. Some useful features of XSLT include:

- Arbitrary result tree structure: During transformation, elements from the source tree can be filtered and reordered, and arbitrary structure can be added to the result tree. Consequently, the result tree can have a completely different structure than the source tree [54].

- Built-in modularity: XSLT provides `xsl:import` and `xsl:include` instructions to allow incorporating XSLT stylesheets programmatically [20]. This is very useful for constructing complex stylesheets modularly.

- Declarative programming: XSLT is a declarative programming language. It provides many programming constructs, such as variables, functions, conditional statements, and iteration statements, for implementing complex transformation logic.
• Extensible: There are two ways to extend XSLT: 1) extend the set of instruction elements used in templates; 2) extend the set of functions used in XPath expressions. The former is often used by XSLT processors to add custom instruction elements. For example, the Redirect extension element in Xalan [52] can be used to redirect output to one or more files. The latter allows adding custom functions written in imperative programming languages (e.g., JavaScript).

The powerful features of XSLT makes it suitable for building tools such as document converters and custom code generators.

Given that XML can be used to create application-specific language, SONAR uses an XML representation of aspects to achieve an AOP language-agnostic notation. An interesting technical advantage to this approach is that it allows SONAR to leverage XSLT and a wide variety of XML-processing tools.

2.3 Management Tools

 Originally known as Java Management Application Programming Interface (JMAPI), Java Management Extensions (JMX) is a standard Java-based technology. It provides a standard means to enable manageability for any Java-based application, service, device or network. JMX defines the architecture with associated design patterns, the APIs and the services for monitoring and management in the Java platform [30]. JMX is protocol-neutral and it allows applications to be integrated with existing management solutions.

Nowadays, JMX is widely adopted in many domains. For example, the Java Virtual Machine (JVM) uses JMX to expose its internal states and statistics (e.g., memory usage, garbage collection statistics) to applications and management solutions. There is a wide range of JMX related tools available. As illustrated in Figure 2.3, JMX consists of four layers:

• Instrumentation layer
• Agent Layer
• Distributed Layer
Figure 2.3: JMX Architecture [28]
• Manager Layer

The following sections discuss each of the above layers in detail.

2.3.1 Instrumentation Layer

The instrumentation layer instruments managed resources which can be applications, services, or devices. The instrumentation of a managed resource is provided by Managed Beans (MBeans). An MBean is a Java object which implements a specific interface and conforms to certain design patterns [30]. MBeans encapsulate resources and expose management interfaces through a JMX agent for remote management and monitoring. There are four types of MBeans: Standard, Dynamic, Open, and Model MBean.

A notification model is defined to allow MBeans to broadcast management events, called notifications. A set of MBean metadata classes are introduced for describing MBeans’ management interface (e.g., attributes, operations and notifications).

2.3.2 Agent Layer

The agent layer defines the JMX agent which consists of the MBean server and agent services. The MBean server is the core component of the JMX agent infrastructure. It acts as a registry for MBeans. In order to be discovered, an MBean must register with the MBean server using a unique key called object name. The MBean server also acts as an intermediary. MBeans in the MBean server are not directly accessible from outside. Instead, they are always accessed via the MBean manipulation operations defined in the MBean server API, i.e., the MBeanServer interface. This is possible because the MBean server stores information describing the management interface of every registered MBean. The following operations are available on MBeans through the MBean server API:

• Management interface discovery
• Attribute value access (read and write)
• Operation invocation
• Notification handling
• Query based on object name or attribute value.

Agent services add management functionalities to the JMX agent. They can be dynamically loaded, unloaded, or updated [30]. Agent services are often themselves MBeans, allowing them to benefit from the management infrastructure. The JMX specification defines the following core agent services: Timer, Monitoring, Dynamic loading and Relation services, as shown in Figure 2.3. They must be provided in any JMX-compliant agent implementation. Developers are allowed to provide additional agent services to add custom management functionalities.

2.3.3 Distributed Layer

The distributed layer consists of protocol adaptors and connectors for remote management. Protocol adaptors and connectors together make the JMX agent accessible to management applications outside of the agent’s JVM.

Connectors allow access to the JMX agent via various communication protocols (e.g., RMI, JMS). The communication involves a connector server in the agent and a connector client in the management application. All connectors provide the same remote interface for management applications to interact transparently with the agent. As a result, management applications can use any connector indifferently, regardless of the underlying protocol [30].

Protocol adaptors provide a view of the JMX agent and MBeans through another management protocol (e.g., SNMP) or web-based GUI (i.e., HTML/HTTP). They adapt the MBean server API into a protocol-specific representation. In this way, management applications can access the JMX agent through their own protocols instead of through a remote representation of the MBean server [30]. In other words, protocol adaptors enable remote access from non-JMX-aware management applications.

The JMX agent can contain multiple protocol adaptors and connectors to allow remote management through different protocols simultaneously.
2.3.4 Manager Layer

The manager layer includes remote management applications which are outside of the JMX agent’s JVM. They communicate with the JMX agent through protocol adaptors or connectors.

In the SONAR model, management tools must include a broad range of support services such as simple, low-level system calls, application-specific customized servers, and high-level standard-compliant environments. Each of these points in the management spectrum requires different levels of integration to be compatible with the overall SONAR approach. Low-level system calls, such as `vmstat` for monitoring memory usage on a Unix system can be deployed with simple shell scripts, whereas Java Management Extensions (JMX) [49] services for application and network management and monitoring require more infrastructure support for dynamic aspect management and state querying through standard management features. This support is detailed further in Section 3.3, and examples of each are provided in Chapter 5.

2.4 Summary

This chapter covered selected background information on the key technologies used in SONAR: AOP, XML/XSLT and JMX. AOP and XML/XSLT together can be used to build language agnostic dynamic instrumentation; JMX can be used for managing the instrumentation operations. The detailed design and usage of these technologies is discussed in Chapter 3.
Chapter 3

Design and Implementation

SONAR’s model is designed to work with a range of AOP support and a range of management tools. Figure 3.1 shows a high-level perspective of this model, where an XML definition is first transformed to a specific, concrete representation in an AOP language and subsequently applied to the points in the execution of the system. This general model allows for aspects to be introduced at any level, from the application to the operating system, and to be available for visualization and management through a range of management tools, from simple system calls to more sophisticated JMX tools. The management interface tools are critical for effective aggregation and filtering of diagnostic data. Through the interface, collections of data should be either consumed or freed effectively, and collection artifacts such as buffer sizes for diagnostics must be easily configured when required.

Simple aggregation functions can allow for the data to be processed at the time of collection, such as calculating averages as provided by DTrace [19]. As highlighted in Figure 3.1, the differentiation of domain independent versus domain specific API for interfacing with tool support determines the degree to which the tool is portable. We envision most low-level tools to be tied to a domain specific API, whereas higher level tools can leverage standardized interfaces. Examples of each are provided in Chapter 5.

It is important to note here that data collection, processing and communication introduce overhead, and sometimes this overhead can be unacceptable and perturb the system being observed. This consequence is commonly known as the observer effect. But it is the effect that SONAR is actually designed to minimize, by allowing
Figure 3.1: SONAR architecture, showing a high-level overview of an XML definition of an aspect. The figure outlines how an aspect can be introduced to crosscut a system and interoperate with associated tool support.

depvelopers to customize selected points more easily and flexibly during the execution of the system to be monitored, and allowing focus to change as information is gathered. This allows for precise collection of data used for system diagnosis, according exactly to the points of interest for the stakeholder.

In one particular manifestation of the SONAR model, dynamic aspects can be generated from XML-based definition files, deployed to applications/frameworks/middleware, and managed through JMX-compatible tools. Management of these dynamic aspects can be further enhanced through a centralized SONAR repository to help prevent a disjointed view of dynamic optimizations in the system. The APIs, upon which the JMX tools manage the aspects, break down into those that are domain independent, such as deployment/removal of aspects, and those that are domain specific, such as system navigation. From there, JMX can be used to visualize and manage the system. The following subsections provide more detail on this particular configuration for SONAR—that is, the combination of dynamic AOP, XML and JMX, respectively.
3.1 Dynamic AOP Integration

To achieve dynamic instrumentation, we chose dynamic AOP since it provides general-purpose, language-level (code-centric) support for augmenting existing systems for various purposes. The common join point model shared by many existing AOP frameworks (e.g., AspectWerkz [8], AspectJ [6], AspectC [5]) provides a solid foundation for implementing instrumentation. This model covers principled execution points in a system written in a number of languages, and thus enables the full range of fine and coarse-grained instrumentation required for comprehensive diagnosis and optimization.

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 and removed from targets. The current implementation of SONAR uses AspectWerkz, a high-performant, lightweight dynamic AOP framework, to provide dynamic AOP on the Java platform.

To mitigate the impact of ad hoc optimizations clouding the overall clarity of the system, SONAR provides a centralized repository of the dynamic aspects currently applied, in order to promote reasoning about their composition in an executing system. Though this is a start, we believe a more comprehensive solution for reasoning about aspect-compositions is required. This is discussed in more detail in Chapter 6 with respect to future work.
Figure 3.2 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.

### 3.2 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. Examples from each of these languages are further explored in Chapter 5. Listing 3.1 shows a sample XML definition for an `httpMonitor` aspect. This aspect essentially monitors executions of the `process` method in the `Http11Processor` class.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sonar>
  <system name="test" start="auto">
    <aspect name="httpMonitor" class="sonar.aspect.MonitorAspect"
      deployment-model="perJVM" manageable="true">
      <advice name="monitorTest(JoinPoint)" type="around"
        bind-to="methodToMonitor">
        <action type="before">
          <![CDATA[
            log("...");
          ]]>
        </action>
        <action type="after">
          <![CDATA[
            log("...");
          ]]>
        </action>
      </advice>
    </aspect>
  </system>
</sonar>
```

Listing 3.1: Sample XML definition file showing an around advice
The core content of every aspect specified in SONAR includes variable/method declarations, pointcut expressions, advice with actions, and parameter definitions. The current schema considered here is based on AspectWerkz’ aspect definition schema, which is similar to those of other existing AOP frameworks. Additional elements of the schema are required for transformation and code generation.

More specifically, line 3 specifies the target system name, and that the aspect is 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 (other options include *perClass*, *perInstance*, and *perThread* as documented in [8]). Line 5 identifies a pointcut (*methodToMonitor*) associated with the execution of the *process* method, and lines 6 through 17 define the functionality (around advice, shown here as a special case of before/after with an automated proceed\(^1\), bound to the pointcut in line 5) to be applied. Line 18 defines the parameters to use for the aspect (details in [8]).

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 Listing 3.1 (lines 9 and 14) is defined outside of the aspect code and would be specific to the target domain. The aspect merely specifies its invocation. It may be implemented as printing to screen, writing to a log or sending to some management console. As a result, the implementation choice of such a method can be made independently from aspect code and therefore, can be customized based on the target system and the target management tool. Further examples of how the use of domain independent and specific APIs can be used for both optimization and navigation are provided in the examples in Chapter 5.

As shown in Figure 3.3, XSLT is used to transform XML definition files into other XML files, such as the aspect definition file for AspectWerkz (Listing 3.2), aspect code (Listing 3.3) and other required source code such as interfaces and helper classes for management purposes. If variable/method declarations and actions in the aspect def-

---

\(^1\)This is a special feature of SONAR, and can be overridden if the *autoproceed* configuration option is set to *false*. 
Figure 3.3: XML definition files are transformed into configuration files and source code in a target language using XSLT and a domain-specific compiler/code generator.
inition are written in domain-specific languages, a domain specific compiler must be used to compile the code into the language used in the target system. SONAR accommodates this kind of heterogeneity by supplying multiple transformers, generating the language-specific aspects, one per target language.

```xml
<!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(*
      <advice name="monitorTest(JoinPoint)" type="around"
          bind-to="methodToMonitor" />
      <param name="..." value="..." />
    </aspect>
  </system>
</aspectwerkz>
```

Listing 3.2: AspectWerkz’ `aop.xml`, generated by transforming the definition file in Listing 3.1

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

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

    Object result = joinPoint.proceed();
    log("...");

    return result;
}
```

Listing 3.3: Java source code containing AspectWerkz-specific code, as prescribed in Listing 3.2

### 3.3 JMX Management

Java Management Extensions (JMX) [49] is gaining momentum as an underlying architecture for Java 2 Enterprise Edition (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 to support both optimization and navigation through standard management features.

JMX can be used as a means to visualize and manage aspects introduced by SONAR. This includes retrieving data from aspects, invoking operations, and receiving event notification. Aspects are themselves MBeans registered with an MBean server. Consequently, aspects can be managed by JMX-compatible tools remotely and/or locally. For example, JConsole, a JMX-compliant graphical tool for monitor-
Figure 3.4: Data from the *MonitorAspect* is visualized as data values change over time, and can be updated in JConsole.

These figures specifically show how the simple *MonitorAspect*, which monitors HTTP requests, database access and JSP service, is visualized and managed in SONAR. Figure 3.4 illustrates how the statistics from three different invocation points monitored by the *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 3.5 shows the operations supported by the *MonitorAspect*: *deploy*, *undeploy*, *displayRecords*, *resetRecords*. The *deploy* and *undeploy* operations are used to manage aspect deployment at runtime and they are shared by all dynamic aspects introduced by SONAR. After being undeployed, advice defined in the *MonitorAspect* are com-
Figure 3.5: Operations (both generic and domain specific) of the *MonitorAspect* are listed and can be invoked manually in JConsole.
pletely removed from all targets. However, the MonitorAspect is still registered with the MBean server. Therefore, it can be accessed by JConsole and can be redeployed by invoking the deploy operation from this management interface (Operations tab, shown at the top of the figure).

The Notifications tab to the right of the Operation tab, can be used to display notifications emitted from MBeans. The MonitorAspect does not have any notification. Thus the Notifications tab is disabled as in the figure. The Info tab displays information about the MonitorAspect.

3.4 Summary

In this chapter, we discussed how SONAR combines dynamic AOP, XML and JMX to meet the key requirements outlined in Chapter 1. The overview of SONAR’s supporting infrastructure discussed in the next chapter gives a more detailed perspective on the ways and means used to collect dynamic information within a system.
Chapter 4

Supporting Infrastructure

This chapter introduces the supporting infrastructure in SONAR for building higher-level services.

In order to cope with the ever-increasing load on systems, both hardware and software technologies have evolved significantly and become more and more sophisticated, i.e., multi-core processor, multi-tier caching, horizontal and vertical clustering, distributed and grid computing. Systems become very complex in order to utilize such sophisticated infrastructure correctly and efficiently, and also to meet various non-functional requirements such as throttling, load balancing or fail-over. In many such systems, multiple distributed and collaborating servers might be involved in serving a single request. Therefore, the overall processing of one request can be split into a tree of sub-processing steps performed sequentially or concurrently in different threads, processes and/or machines. The structure and content of this tree are quite dynamic and can differ significantly on a request-to-request basis. For instance, if the same request is made twice to a system, it can be served by two completely different set of servers due to load balancing. Likewise, when several requests are received at the same time, some might be processed successfully while others might fail due to failures in some parts of the system.

In short, the scale, complexity and dynamic nature of large distributed systems make it difficult to perform certain diagnostic and analytic tasks, such as root cause analysis of failing requests. Therefore, we believe it is crucial for SONAR to provide an infrastructure facilitating system-wide end-to-end tracing on a per-request basis. The following are the detailed requirements for such infrastructure:
• Ability to track a particular request and its descendant sub-requests throughout a system regardless of the processing model (sequential, concurrent or a mixture of both).

• Ability to specify which request to be tracked programmatically.

• Ability to identify each request along with its descendents uniquely. The parent-child relationship between a request and its descendents are available for post-processing analysis (i.e., reconstruct the processing step tree).

• Ability to tag artifacts (i.e., log messages) so that they can be correlated with the corresponding processing step.

In order to achieve these, SONAR provides three supporting facilities: Flows, Probes and Logs which are discussed in detail in the following sections.

4.1 Flows

Flows are the fundamental facility in SONAR for tracking requests in distributed systems. More specifically, they facilitate tracking and replaying the processing steps of a particular request across logical (i.e., application-level components, threads) and physical (i.e., machines) boundaries throughout or in parts of a system. The design of Flows is based on the following assumptions/observations:

• Timestamps are not a reliable mechanism for ordering events occurred in distributed environments. This is because the machine clocks may not be properly synced or not to an acceptable accuracy.

• Throughout its processing lifecycle, a request can be handled in multiple threads across different machines in a sequential and/or concurrent manner.

• While processing a request, a thread can pause it, switch to process another one, and then resume.

Terminology:

• Flow: a flow represents the processing of a particular request within a system. As a request and its sub-requests are processed, the corresponding flow can span
across multiple threads and/or machines in the form of sub-flows as defined below. Its data structure and states are discussed in detail in the next section.

- **Sub-flow**: a sub-flow is a flow triggered by or split from another flow. It represents a specific step in the overall processing of its parent. A flow can have multiple sub-flows which can further trigger or split into their own sub-flows. The child-parent relationship between a sub-flow and its parent merely imply *caused-by*. It does not imply anything on processing model, timing or scheduling. Therefore, a sub-flow can occur sequentially or concurrently to its parent and also its siblings—the other sub-flows from the same parent. It can occur in the same or a different thread/machine than its parent and siblings. Furthermore, it is not necessary for a child to be completed before its parent. For instance, the parent flow can fire an asynchronous event and then terminate immediately. The sub-flow associated with the event handling might occur before or after the parent flow completes. Moreover, the creation order of sub-flows does not imply the processing order.

- **Propagation**: the process of propagating flow states along with requests from one part of the system to another (local or remote)—discussed in detail in Section 4.1.3.

- **Branching**: the process of creating a sub-flow in a flow—discussed in detail in Section 4.1.4.

The following subsections discuss flow data structure and handling.

### 4.1.1 Data Structure

As shown in Figure 4.1, *Flows* consist of four main classes: *Flow, FlowStore, FlowManager* and *FlowListener*.

- **Flow**: has the following properties:
  - *Id*: an identifier which uniquely identifies a flow and its descendant sub-flows. When a request enters a flow-enabled system, a unique id is generated and used throughout itself and its descendents’ lifecycles. It can also be provided by the clients who made the request, such as external system,
Figure 4.1: Flow and related classes
system administrator or automated tools. The exact mechanism for providing flow id is discussed in detail in Section 4.1.3. For generated ones, SONAR ensures the uniqueness on a per-machine basis. Global uniqueness is achieved by combining flow id with machine identifier. For provided ones, it is the clients’ responsibility to ensure such uniqueness.

– **Settings**: contains the configurations for a flow and its descendant sub-flows, i.e., verbose mode, flow artifact handling (e.g., broadcast the intercepted log messages) as shown in Figure 4.2. This allows different flows to be handled differently. Similar to flow id, settings can be provided by clients on a per-request basis. If no settings are provided, the default settings would be applied. The default settings for flows are managed by FlowManager.

– **Parent stamp, local depth and local sequence number**: internal data structure for tracking and post-processing. They allow each sub-flow to be uniquely identifiable against the others triggered by the same root request. These are maintained internally by SONAR and therefore cannot be provided by clients. Their definition and usage are discussed in detail in Sections 4.1.3 and 4.1.4.

- **FlowStore**: allows flow states captured in the properties discussed above to be saved and retrieved in a thread-safe and efficient manner. It is used for flow states propagation as discussed in Section 4.1.3.

- **FlowManager**: provides flow management operations, which are covered in detail in the next section. It also provides flow listener (see below) registration and de-registration.

- **FlowListener**: the interface for implementing listeners to receive flow lifecycle events, i.e., flow begins and ends. By registering listeners with FlowManager, we can inject custom actions into flow lifecycle. Flow listeners are registered against and managed by a FlowManager. Once registered, a flow listener would be invoked when an event occurs from any flow managed by the same flow manager.
Moreover, since flow listeners are invoked in the request processing threads, the actions performed in flow listeners must be quick and non-blocking. In other words, flow listeners should only perform minimum non-blocking actions and, if necessary, schedule blocking and/or time consuming actions to be executed in other thread(s).

The flow handling process is about maintaining, manipulating and transferring of flow states using the data structure discussed in this section. Currently, it is divided into three functional aspects: management, propagation and branching. They are described in the following subsections, respectively.

### 4.1.2 Flow Management

Flow management accommodates complex processing models existing in today’s distributed systems. It is designed as, while being actively processed, a flow is associated with one and only one thread—the *current thread*; if a request is processed concurrently in more than one thread, each thread must have a unique flow associated with it.

Terminology:

- **Current thread**: a thread which is currently running.
- **Local depth**: the number of activations minus the number of deactivations on a flow since it enters a server/VM/thread. When a flow is created, its local depth is set to 0.
- **Current flow**: a flow which is associated with the current thread. Its states are stored in the current thread’s thread-local variables.
- **Active flow**: a flow is active if and only if its local depth is greater than 0.

In the current implementation, flow states are stored and accessed on a per-thread basis as thread-local variable(s). This approach is inspired by how existing logging frameworks support application contextual information when logging messages. It

---

1A static or global variable refers to different memory locations when being accessed by different threads.
Figure 4.3: Flow states and operations

allows easy and efficient flow state update and retrieval. Section 4.3.3 demonstrates how flow-aware log interceptors can be easily built on top of this.

Flow states are managed by FlowManager and not directly accessible to applications. Therefore, applications invoke FlowManager’s flow operation methods to manipulate the current flow states. As illustrated in Figure 4.3, there are five state manipulation operations:

- **initialize**: to clear the states of the current flow. Its local depth is set to 0 to indicate that it is inactive.
- **begin**: to activate the current flow by incrementing its local depth by 1.
- **save**: to save the states of the current flow into a FlowStore or a string.
- **resume**: to restore the flow states from a FlowStore or a string into the current flow and then calls begin. If specified, this resets the local depth to 1.
- **end**: if the current flow is active, decrement its local depth by 1. If the local depth is down to 0 after decrement, re-initialize the current flow by calling initialize. Generally, this is invoked in the same method body where begin is invoked.

In addition, FlowManager provides operations for retrieving the states of the current flow, managing default settings for flows, and FlowListener registration/de-registration on a per FlowManager instance basis.

### 4.1.3 Flow Propagation

Flow propagation is the process of propagating flow states along with requests from one part of the system to another (local or remote). As described in the previous
section, flows are managed on a per-thread basis in the SONAR implementation. Therefore, the flow propagation is about how to propagate flow states across thread boundaries (local or remote).

In order to cross thread-boundaries, a flow can be in two forms: an object of type `Flow` as described in Section 4.1.1 or a string. A flow object can be encoded into and decoded from the string form. The string form consists of `id`, `settings` and `stamp` as illustrated in Figure 4.4. Note that it does not contain the local data (i.e., local depth, local sequence number) as in the object form.

As described in Section 4.1.2, when a flow-attached request is being processed in a thread, the attached flow’s states are stored in that thread’s thread-local variable(s). Therefore, a flow is considered active when that thread is actively processing the request which it attaches to. Flow propagation happens when the request processing is either paused-and-then-resumed in the same thread or transferred to another thread (local or remote). The three different scenarios of flow propagation are:

- **In-thread**: this is a pause-and-then-resume case. It is common in threads which can process multiple requests at the same time. Strictly speaking, a thread can only actively process at most one request at any given time, with respect to the processor to which it is scheduled. However, these threads can pause the processing on one request and switch to process another one. In other words, while processing a flow-attached request, the thread can pause the processing in order to perform other tasks (e.g., process other requests), and then later resume the processing of the unfinished request. Since the flow states attached with a request are stored in thread-local variables and the other requests processed by the same thread in between the pause and resume can have flows attached as well, we cannot rely on using thread-local variables to store flow states over a pause-and-then-resume period. Thus, flow states have to be saved before the pause and then restored them after resume.

![Figure 4.4: Flow in string form](image)
The flow states can be saved either to a FlowStore object or to a string. The latter is more expensive due to string construction and parsing costs, and therefore it is not recommended in this scenario. Before the pause, the current flow states are copied out from thread-local variables and saved into a FlowStore object. The thread-local variables are cleared afterward so that they do not affect the processing of later requests. Prior to resuming processing of the request, its corresponding flow states must be restored from the FlowStore object back into the thread-local variables. Once the request is completed, its associated flow states are cleared from thread-local variables.

Furthermore, it is recommended to attach a FlowStore object to a request object so that the corresponding flow states can be stored and retrieved easily. This is usually done by either explicitly declaring a FlowStore field in the request object class or using AOP’s mixin mechanism. Adding flow stores directly into request objects eases memory management since a flow store object along with the saved flow states will automatically become eligible for garbage collection whenever the corresponding request object is no longer referenced. However, the drawback of this approach is that it can cost more memory due to the extra field in the request class, especially in systems with a large number of small request objects. Moreover, this limits a request object can only be processed by one thread at a time. In situations that multiple threads can process a single request object concurrently, instead of having one flow store, the request object needs to have a map of flow stores keyed by thread ids.

- **Across local threads**: local means the threads are in the same VM. Propagation across local threads is done in a similar way as in-thread propagation. Nevertheless, the differences are that the target thread might handle request objects of a different class, and upon resuming the flow in the target thread, the local depth is reset to 1. In such case, flow states saving and restoring should be done on a new request object. For example, a web component receives a web request which requires saving data asynchronously to database. Then it creates a data request object which is put on the work queue of the data access component running in a different thread. If the current flow is active, the web component would save the flow states into the data request object, instead of the web request object, before putting it on the queue. When the data request object is
pulled off the queue in the data access thread, the saved flow states would be restored into the data access thread’s thread-local variables and the *local depth* is reset to 1.

- **Across remote threads**: *remote* means the threads are on different VMs. To pass a request object across VM boundaries, various communication protocols can be used. As a result, flow propagation across remote threads is different from across local threads in terms of how flow states are saved and restored. The exact mechanism is communication protocol dependent. For instance, if RMI/TCP is used for remote request, embedding serializable `FlowStore` objects directly into request objects can work. If SOAP/HTTP is used, instead of using `FlowStore` objects, flow states can be encoded into a string and added as HTTP header, HTTP request parameter, or SOAP envelope header. On the receiving end, a corresponding protocol specific filter, e.g., a servlet filter checking some HTTP header, can detect, decode and resume the flow in the remote VM. When sending a response back, the flow states can be transmitted in the same way as sending a request.

### 4.1.4 Flow Branching

As discussed in the previous sections, systems can have a complex processing models, i.e., a request can recursively trigger sequential or concurrent sub-requests which can be processed in different threads and/or machines. The flow propagation mechanism addresses how to propagate flow states from one thread to another (local or remote). However, without other handling, it would work only in the simplest strict-sequential processing model. In order to address this problem, SONAR introduces a flexible flow branching mechanism.

Flow branching is the process of spinning off a new sub-flow from a flow/sub-flow. By using this mechanism, a flow can branch into multiple sub-flows and each sub-flow itself can be further branched. In addition, sub-flows can occur sequentially and/or concurrently to each other and to their parent. SONAR ensures that each sub-flow is identifiable and traceable. Identifiable indicates each sub-flow can be uniquely identified against the others triggered by the same root flow. Traceable means each sub-flow contains previous branching information which can be used to identify its parent and find out about the relationships (i.e., processing order, sequential or con-
Figure 4.5: Flow stamp and branching

current) with its siblings—other sub-flows from the same parent. A tree can be used as an analogy for flows, with the trunk as the root flow and tree branch(s) as sub-flow(s). From any branch, there is a unique path of parent branch(s) leading to the trunk. The path is equivalent to the branching information stored in each sub-flow. With this information, we can reconstruct the complete processing sequence of a flow and its descendents. Moreover, the branching information can be used to tag the artifacts (e.g., logs) collected during request processing. By doing this, tools (e.g., request path visualizer, log viewer) can associate the artifacts to the corresponding processing steps.

The basic technique in flow branching is stamping. A stamp stores the branching information. As illustrated in Figure 4.5, suppose a flow, with stamp *, branches into n immediate sub-flows. Suppose each sub-flow is assigned with a unique number in the range 1..n, and then its stamp is set to *: {1..n}. If any of the immediate sub-flows further branches into m children, each of them would have a stamp *: {1..n}: {1..m}. Furthermore, in order to differentiate local (i.e., within the same VM) branching from remote (i.e., across VMs) branching, SONAR uses two delimiters: ‘:’ for local branches and ‘;’ for remote ones. The only reason for such distinction is to make post-processing easier.

The two properties in the Flow class are used for stamping: local sequence and stamp. The local sequence is an ever-increasing number managed on a per-flow/sub-flow basis. Upon branching, a flow assigns its local sequence to the newly created sub-flow. When a flow/sub-flow is first created, its local sequence always starts from 1 and it is incremented on every branching. A flow’s stamp is the list of local sequences assigned by its ancestors. As in Figure 4.5, a stamp is in the format *: {1..n}:.. The following discusses the exact manipulation on local sequence and stamp upon flow
Asynchronous branching:

- **Asynchronous**: a sub-flow is asynchronous to its parent if its execution would not block its parent. For example, upon receiving a request, a server sends out an asynchronous event and then proceeds with the processing. The processing in the event listener(s) is an asynchronous sub-flow. As illustrated in Figure 4.6, before branching, the local sequence, \( s \), of the parent is incremented by 1, \( s+1 \). It then is appended to parent’s stamp, \( * \), to form the sub-flow’s stamp, \( *:s+1 \). After branching, parent continues using local sequence \( s+1 \)—which indicates that the processing of parent after branching is parallel to the sub-flow.

Synchronous branching:

- **Synchronous**: a sub-flow is synchronous to its parent if the parent’s execution is blocked until the child terminates. For example, in order to serve a request,
a server makes a blocking call to another server and waits for the response. The processing in the second server is a synchronous sub-flow. As illustrated in Figure 4.7, the sub-flow would have the same stamp as the asynchronous case. However, the local sequence of the parent after branching is incremented once again—which indicates that the processing of parent after branching is performed after the sub-flow completes.

The local sequence is also incremented when either `begin()` or `end()` is invoked on a flow. Even though there is no branching involved, these calls still indicate the entering and exiting of some specific code block of interests—surrounded by a `begin()` and `end()` pair. Since the local sequence, along with other flow states, is used to tag collected artifacts, incrementing on both calls allows identifying and separating the artifacts generated in the surrounded code block from the rest. This even allows separating repeated executions of the same code block, e.g., looped calls to a method surrounded by a `begin()` and `end()` pair.

4.2 Probes

The Flows facility discussed in the previous sections addresses how to track a request and its descendant sub-requests throughout a system. Without any other facility, flows need to be explicitly activated by clients (e.g., other servers, system administrators or tools). This section introduces the Probes facility which supports criteria-based flow activation. The data structure, the management and process, and how to integrate it with flows are described in the following subsections.

4.2.1 Data Structure

As shown in 4.8, the probes facility consists of the following members:

- **Probe**: a probe is bound to a particular type and it allows applications to report data objects of that type via the `report()` method. There are two reasons behind this type binding: first is for easy categorization (e.g., probes accepting exceptions would be bound to target exception types) and management (e.g., search for probes by data type); second is to make it less error-prone to use. A probe can be in either activated or deactivated state. The methods for manipulating probe states are only visible to the `ProbeManager` (see below). The details on
Figure 4.8: Probe and related classes
these states are discussed in Section 4.2.2. In addition to the data object, the report() method takes another parameter, reporter, which is the identity of the object reporting the data object. At the end of a report call, a probe creates and returns a SessionHandle (see below).

- **Session**: a session represents the processing between a report call on a probe and the close call on the session handler returned by the report call (see below). It includes the criteria matching and session listener(s) invocations, and the execution of application code in between the report and close pair. A session object, which is immutable once created, holds the references to the probe, the reported data and the reporter. These referenced objects can then be accessed by CriteriaMatcher, SessionListener and application code (via SessionHandle).

- **SessionHandle**: session handles can only be created and obtained via invoking the Probe.report() method. A session handle holds the reference to the associated session and provide the operation to close it—the close() method. The handling details upon closing is discussed in Sections 4.2.2 and 4.2.3.

- **CriteriaMatcher**: an interface for criteria matching providers. Same as probes, a criteria matcher is bound to a specific type and it can only be registered with probes bound to compatible types. For example, if a criteria matcher is bound to a type T, it can be registered only with probes bound to T or any of its descendant types. Criteria matchers are registered with and de-registered from probes via ProbeManager’s addCriteriaMatcher() and removeCriteriaMatcher methods, respectively.

- **SessionListener**: an interface for implementing listeners to receive session events, i.e., opening or closing a session. Same as criteria matchers, a session listener is bound to a specific type and it follows the same type restriction when registering with probes. Session listeners are registered with and de-registered from probes via ProbeManager’s addSessionListener() and removeSessionListener() methods, respectively.

- **ProbeManager**: provides probe management operations i.e., probe creation, activation and deactivation, and searching. It also provides criteria matcher and session listener registration and de-registration. In addition, it internally keeps track of the usage (i.e., execution count) and statistics (i.e., timing) of
probes, criteria matchers and session listeners. This data provides insights into how various components inside the Probes facility perform.

4.2.2 Probe Management and Process

As discussed previously, each probe is bound to a specific type and managed by the probe manager. The probe manager provides probe lookup via the `getOrCreateProbe()` method which creates a probe if no match is found. This lookup approach provides easy and uniform access to probes without passing them around, and it also facilitates lazy-initialization of probes.

Probes are managed and accessed either by type or by name. In the former case, the probe manager maintains the one-to-one mapping between a type and a probe. In the latter case, instead of a type, a probe is mapped to a name. A named probe is still bound to a specific type but it can only be accessed via its name specified upon the first lookup. This addresses the need for more than one probe to be bound to the same type.

The probing process includes the following steps:

- **Reporting data to a probe and obtaining a session handle:** upon the report call, a probe first checks whether it is activated. If not, it simply returns a dummy session handle. If it is, the probe creates a session object and begins the criteria matching process by invoking the registered criteria matchers’ `match()` method with the session object as parameter. If any criteria matcher returns `false`, the criteria matching process ends immediately and a dummy session handle is returned. In the case of all criteria are met, the probe invokes the registered session listeners’ `sessionOpened()` method, creates and returns a session handle.

- **Closing the obtained session handle:** upon the close call, a dummy session handle returns immediately. A not-dummy session handle notifies the registered session listeners by invoking their `sessionClosed()` method in the reverse order as in which their `sessionOpened()` methods were called.
Figure 4.9: Probe and flow integration
4.2.3 Probes meet Flows

The design of flows and probes are discussed separately in previous sections. This section discusses how probes are integrated with flows. As illustrated in Figure 4.9, there are two cases:

- **Triggering**: upon the report call, an enabled probe detects no active flow in the current thread. It then performs the criteria matching process as described in the previous section. If all criteria are matched, a new flow with a generated id is created and started (via calling the `FlowManager.beginFlow()` method). The returned session handle calls the `FlowManager.endFlow()` method upon closing. If the probe is disabled or any criteria fail to match, a dummy session handle is returned.

- **Passing-through**: upon the report call, an enabled probe detects an active current flow. It then skips the criteria matching process, and calls the `FlowManager.beginFlow()` method. Same as in the triggering case, the returned session handle calls the current flow’s `end()` method upon its `close()` call.

`Probe.report()` and `SessionHandle.close()` invoke the `begin()` and `end()` methods on the current flow, respectively. Their usage follows the same pattern as flows’, as illustrated in Listing 4.1 and 4.2.

```java
boolean beginFlow = ... // Criteria matching

if(beginFlow) {
    FlowManager.beginFlow(); // Flow begin
}

try {
    // Process
} finally {
    if(beginFlow) {
        FlowManager.endFlow(); // Flow end
    }
}
```

Listing 4.1: Flows usage example
SessionHandle handle = probe.reprot(some_app_data); // Non-null
    handle

try {
  // Process
}
finally {
  handle.close();
}

Listing 4.2: Probes usage example

4.3 Logs

In the following subsections we first briefly establish the role of logging in modern systems, visit existing frameworks, and detail an important part of SONAR’s infrastructure, flow-aware logging.

4.3.1 Why logs?

Logging is a common practice in most, if not all, non-trivial systems. Events occurred in various parts of a system can be logged in plain text or object form. Logs are usually persisted to some non-valotile storage (i.e., file, database). Commonly, some contextual information (such as time stamp, log severity level, location of the corresponding log statement in source code) is logged along with log messages. Logging can be used for various purposes:

• Record keeping, analysis and reporting: Since logs can be persisted along with contextual information to some durable storage, logging can be used as a record keeping mechanism to record any event of interest, e.g., every user transaction. Many analysis and reporting tasks are time consuming and/or requiring more information than what is available to individual components in a system. Therefore, they are usually done in a post-process fashion. Various data stored via logging can later be consumed by log analyzers for analysis and reporting purposes. For example, if the process time is logged for each user transaction, a log analyzer can calculate the system-wide process time distribution by analyzing the logs from all the servers processing user transactions.
• **Debugging**: Debugging using a debugger has several shortcomings. First, it usually requires the target application/VM to run under debug mode which introduces significant cost in time and processing, and possibly requires application/VM restart. Second, it usually requires prior knowledge of what is wrong and where to set breakpoints, and also manual control (i.e., set/clear breakpoints, stepping). Last, most debuggers only work in a single-application/VM scope. Due to these shortcomings, debuggers have very limited use outside of development environments. Logging can be used as a simple and lightweight technique to assist debugging. The occurrence of logs indicates the execution of the code blocks to where the corresponding log statements belong. Since logs are usually time-stamped, their temporal order can indicate the execution order of the corresponding code blocks within the context of a single clock-provider (typically a machine). Furthermore, developers can write out any data to logs for debugging purpose, e.g., method parameters, intermediate results, enter/exit a code block. Moreover, logging usually requires less prior-knowledge than debugging and almost no manual control. It introduces predictable and controllable cost on system resources because log statements are statically defined in code and they can be filtered out using some filtering mechanism (see Section 4.3.2). Unlike debuggers which only work in a single-application/VM scope, logging works naturally in distributed systems. Logs from distributed servers can be correlated and analyzed basing on some contextual and/or non-contextual data carried in them. Therefore, logging is suitable for both development and production environments.

In summary, logging is a crucial part of many systems. It provides detailed information about the states of a system. Logs generated during request processing are very important artifacts for post-processing analysis. Thus, it is crucial for SONAR to be able to collect logs on a per-request basis, in order to understand the inner workings of a system when processing a particular request.

### 4.3.2 Existing Logging Frameworks

Many logging frameworks exist for different programming languages/platforms (e.g., SLF4J [50] for Java). They abstract away common concerns and tasks in logging, and provide simple and uniform APIs for configuration, usage and extension.
They greatly simplify logging in various kinds of systems. However, the following shortcomings are common in these frameworks:

- **Limited control over log filtering:** Log statements are statically defined in code. At runtime, executing log statements introduces significant costs in time, processing (i.e., constructing and writing out logs), storage (i.e., disk space) and possibly network bandwidth (i.e., transmitting logs over the network to a remote log server). In order to control these costs, most logging frameworks provide a filtering mechanism based on log severity levels, such as *ERROR*, *WARN*, *INFO*, *DEBUG* and *TRACE*. Each log statement must specify its severity level and, at runtime, the underlying logging framework determines whether to execute the statement basing on a filtering threshold. For instance, if the filtering threshold is set to *INFO*, log statements with *INFO* or more severe levels (i.e., *WARN*, *ERROR*) would be executed, while the ones with less severe levels (i.e., *DEBUG*, *TRACE*) would be skipped. Furthermore, in order to provide finer-grained control over log filtering, many logging frameworks support package/class-level filtering threshold. Each package/class can have a separate threshold applied to all the log statements within it. Typically, these thresholds are defined in some configuration file and loaded during system startup. Some frameworks even allow changing them at runtime.

This simple and straightforward filtering mechanism allows log statements to be executed or skipped depending on how severe they are and where they are located. However, it does not support per-request filtering because it is merely based on static aspects (i.e., hard-coded severity level and static code structure) and has no notion of runtime context. In other words, filtering cannot be done on a per-request basis because it cannot differentiate a request from other requests. In order to gather all the log messages which are caused by a particular request and of a certain level or above, the filtering thresholds for all the code in the possible execution path(s) must be set to the appropriate level(s)—equal to or less severe than the interested level. For example, to diagnose a failing request, we need to gather all the generated log messages with *DEBUG* level or above. Hence, the filtering thresholds for all the packages/classes in the possible execution path(s) must be set to *DEBUG* or *TRACE* level prior to making the request. This is nearly unacceptable to any production environment with non-
trivial traffic because of the consequent significant costs.

- **Limited logging context scope**: As mentioned above, applications often need to add contextual information to logs for various purposes (e.g., analysis and reporting). Explicitly writing out contextual data in each log statement is a repetitive and error-prone task. More importantly, it is quite a challenge to make contextual data accessible to all the related log statements in an efficient and thread-safe manner. In order to relieve developers from handcrafting the above, some existing frameworks (e.g., Apache Log4j [3]) support runtime logging context via Nested Diagnostic Context (NDC) and/or Mapped Diagnostic Context (MDC) [42]. With these mechanisms, arbitrary objects can be added as contextual data and get written out to logs transparently. More specifically, once an object is added to the logging context, it is transparently available to all the following log statements until being removed. This eliminates the need for explicit passing around and writing out context data. However, the existing implementations of NDC and MDC store the logging context per thread and do not provide any propagation mechanism. As a result, context data are restrained to the thread in which they were added to the owning logging context. In order to propagate across thread/VM boundaries, explicit passing is still required. In other words, the scope of logging context in existing logging frameworks is limited by thread/VM boundaries. This severely limits their usage in multi-threaded and/or distributed systems.

- **Lack of support for log identification and correlation**: Existing frameworks do not provide a generic way for log identification and correlation which are commonly required for diagnostic and analytic tasks. As a result, developers have to manually maintain (i.e., create/retrieve, propagate) some identification data (e.g., user id, request id) and inject them into log messages. This manual approach is tedious and error-prone. More importantly, without a uniform identification scheme, it is difficult to identify and correlate logs in a generic way. Furthermore, without a generic propagation mechanism, some components in a system might not have access to the identification data due to technical and/or security constraints. Moreover, different correlation interests might require different identification data. For example, correlating logs on a per-user basis requires the presence of user id in log messages, while per-request correlation requires request id. With this manual approach, changing identification data
usually requires non-trivial code change.

In short, existing logging frameworks lack the support for collecting logs on a per-request basis which is crucial to fulfill SONAR’s requirements discussed at the beginning of this chapter. Therefore, we introduce SONAR’s solution to this problem in the next section.

### 4.3.3 Flow-aware Logging

In order to address the limitations discussed in the previous section, we decide to roll out a solution which can perform logging both in the traditional way and also on a per-request basis. More specifically, if no active flow is detected while processing a particular request, it does exactly the same as existing logging frameworks—filtering log statements by comparing their statically defined severity levels against the package/class-level thresholds. However, when active flow is found, the flow-specific threshold is used instead of package/class-level ones. As illustrated in Section 4.1.1, the flow-specific threshold is stored in the flow setting bits which are specified upon the creation of the root flow and shared by all descendant sub-flows.

Instead of re-inventing the wheel and introducing a complete logging solution, we choose to extend existing logging frameworks by injecting the flow-aware behavior into existing logging frameworks. Another reason for this approach, compared to develop a new logging framework, is to facilitate easy adoption by applications already using existing ones. Basically, this requires augmenting every logging operations (i.e., `info()`, `debug()`) with flow-aware behavior. The exact mechanism to achieve this is logging framework specific. Since SONAR uses SLF4j to perform its internal logging, we will take SLF4j as an example to illustrate how this can be done.

SLF4j is a simple facade or abstraction for various logging frameworks, e.g., Apache Log4j, allowing the end user to plug in the desired logging framework at deployment time [50]. With SLF4j, logging is typically performed in the following steps: 1. application calls a static logger factory to obtain a logger instance for a specific class/name; 2. perform further logging operations via the returned logger. Based on this, we will discuss how SONAR’s flow-aware logging works with SLF4j (as logging facade) and Apache Log4j (as logging provider) step-by-step in the following sections.
Figure 4.10: Flow-aware logger and factory
4.3.3.1 Logger Creation

As shown in Figure 4.10, SLF4j provides a clean abstraction (i.e., LoggerFactory class, ILoggerFactory and Logger interfaces) which allows easy customization and extension. Basically, SONAR’s solution is built as a wrapper (i.e., SonarLoggerFactory and SonarLogger) to SLF4j’s adapter for Apache Log4j (i.e., Log4jLoggerFactory and Log4jLogger). The runtime discovery mechanism provided by LoggerFactory and other utilities classes in SLF4j allows SonarLoggerFactory to be discovered and then used in place of Log4jLoggerFactory. This is done by simply putting SONAR’s jars ahead of SLF4j’s in classpath. This approach allows SONAR’s logging to be easily plugged into or removed from application without code change. Figure 4.11 illustrates the method call sequences for application to obtain a logger:

1. Application calls LoggerFactory’s getLogger() method

2. LoggerFactory forwards the call to SonarLoggerFactory which in turn delegates to Log4jLoggerFactory.

3. Log4jLoggerFactory finds or creates a Log4jLogger instance which is then returned to Log4jLoggerFactory.

4. Log4jLoggerFactory wraps the Log4jLogger instance into a new SonarLogger instance which is then returned to LoggerFactory.

5. LoggerFactory returns the SonarLogger to application as a generic Logger.

Figure 4.11: Logger creation with SONAR
4.3.3.2 Logging Operation

There are two types of logging operations on a logger:

- Check if a certain severity level is enabled (e.g., `isDebugEnabled()`).
- Log a message with a specific severity level (e.g., `debug(msg)`).

A typical log statement calls the latter directly or if the log message is costly to construct, surrounds it with an `if` statement checking the result from the former. Here we will focus on the latter, since in most implementations, the latter itself would also call the former in order to prevent misuse.

Logging operations are where the severity-level based filtering is applied. Therefore, they are the place to inject and perform flow-aware behaviours. As shown in Figure 4.10, `SonarLogger A` implements the `Logger` interface and holds the reference to `Log4jLogger B`. In other words, A acts as the flow-aware wrapper around B. As discussed previously, upon a call to `LoggerFactory.getLogger()`, `SonarLoggerFactory` wraps B into A. Once returned, application performs all logging operations against A. Figure 4.12 illustrates A’s different behaviours depending on the current flow states obtained from the `FlowManager`:

- When there is no active flow associated with the current thread (as `FlowManager.isFlowActive()` returns false), A simply delegates all the logging operation calls straight to B. B would apply the package/class-level filtering threshold as seen in Section 4.3.2.

- When the current flow is active, A would override B’s package/class-level threshold with the flow-specific threshold specified in the flow setting bits (bit 3 to 5) and perform the actions according to the other settings. Currently, there are two actions (`BROADCAST` and `PROMOTE`) which can be controlled by setting the corresponding flags in flow settings, respectively.

If `BROADCAST` flag (bit 1) is set, A would batch up and broadcast all the intercepted messages with severity level higher than or equal to the flow-specific threshold. The destination address and port are defined in a configuration file.
Figure 4.12: Flow-aware logging
If PROMOTE flag (bit 2) is set, $A$ would promote messages with low severity levels (lower than the package/class-level threshold as in $B$) to higher level in order to pass $B$’s filtering. In the example shown in Figure 4.12, application tries to log a message with DEBUG level which is equal to the flow-specific threshold but lower than the package/class-level one (INFO). If $A$ forwards the message to $B$ using the original level DEBUG, it would get filtered out. To work around this, SONAR allows severity level promotion. Basically, it increases message’s severity level in order to pass package/class-level threshold. The original level information is preserved by converting it to text and inserting the result to the message itself. For the above example, $A$ would promote the level from DEBUG to INFO, and add text /DEBUG/ to the message.

Furthermore, in addition to application logs, SONAR’s internal events/logs can also be collected in the same way. This is because SONAR uses SLF4j for its own logging purpose. The VERBOSE flag (bit 0) in the setting bits controls SONAR whether to log certain events (i.e., flow lifecycle events) via SLF4j—which in turn would be intercepted by the flow-aware logger. These events/logs are useful for analyzing request processing steps, especially when the corresponding code blocks contain no logging statement.

In summary, SONAR’s flow-aware logging solution adds flow-awareness to existing logging framework in a non-invasive way. It allows log filtering threshold and other actions (i.e., broadcast intercepted log messages) to be configured on a per-flow/request basis. As a result, it facilitates collecting log messages of any level regardless the package/class-level thresholds set in the underlying logging framework.

4.4 Summary

In this chapter, we discussed the supporting infrastructure provided by SONAR: Flows facilitate tracking individual requests in distributed systems, while Probes provide a flexible criteria-based flow activation mechanism. We also covered how to intercept and collect log messages on a per-request basis. A case study of using this infrastructure follows in Chapter 5.
Chapter 5

Case Studies: Optimization and Navigation with SONAR

In this Chapter, we assess the feasibility of the SONAR model by applying it in four different scenarios: (1) at the level of an operating system optimization, with rudimentary dynamic analysis tools supported by the OS, (2) at the level of a virtual machine monitoring strategy for garbage collection, introducing rudimentary analysis tools to the system, (3) at the level of a simple web-based Java application, this time with JMX support and additionally utilizing the supporting infrastructure provided by SONAR, and finally (4) at the level of a web-based application that runs on top of the .NET framework. These scenarios provide a proof-of-concept prototypes for optimization and monitoring functionality at their respective levels of the software stack. Additionally, they allow us to start to identify some of the costs associated to applications that may adopt SONAR for these purposes.

5.1 Operating System Optimization

We studied and presented the impact of aspects on operating system code previously [14]. Here we demonstrate how these same operating system aspects, refactored from the original source code, can be incorporated into the SONAR model. We consider an optimization for accessing files that have been mapped into main memory. This prefetching aspect performs a read-ahead from disk in order to reduce access latencies. We use SONAR’s AspectC transformer to realize the aspect, which is then statically incorporated into the build of the system. We rely only on the most simple,
low-level, pre-existing system diagnostic tools in order to loosely monitor impact of the functionality introduced. As the AspectC prototype does not support dynamic aspects, the instrumentation cannot be modified at runtime. At this level, these management tools at least provide a starting point for cost-effective system monitoring and management. In future work we plan to explore facilities for aggregating and filtering data provided by these pre-existing tools.

An effective system optimization found in several versions of FreeBSD [23] concerns prefetching, a heuristic used to reduce disk latencies by attempting to bring pages into memory in advance of explicit requests for them. For example, if sequential access to a file is detected, a page fault will result in a disk request not only for the missing page, but also for several pages in advance of this page, in anticipation of future accesses. Here we consider an aspect associated with this functionality for FreeBSD v3.3. SONAR’s XML specification for the prefetching aspect is shown in Listing 5.1, and its subsequent manifestation in AspectC is shown in Listing 5.2.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sonar>
  <system name="file">
    <aspect name="filePrefetch"
      class="sequential_mapped_file_prefetching"
      language="AspectC">
      <import>...
    </aspect>
    <pointcut name="fault_path" params="vm_map_t map"
      expression="cflow(execution(int vm_fault(map, vm_offset_t,
          vm_prot_t, int)))" />
    <pointcut name="ffs_read_path" params="struct vnode* vp,
          struct uio* io_info, int size, struct buff** bpp"
      expression="cflow(execution(int ffs_read(vp, io_info, size,
          bpp)))" />
    <advice type="before">
      <pointcut params="vm_map_t map, vm_object_t object,
          vm_page_t* pagelist, int length, int faulted_page">
        <expression>
          <![CDATA[
            execution(int vnode_pager_getpages(object, pagelist,
              length, faulted_page))
            && fault_path(map)
          ]]>
      </advice>
    </aspect>
  </system>
</sonar>
```
<pointcut params="vm_object_t object, vm_page_t * pagelist, int length, int faulted_page" expression="execution(int ffs_getpages(object, pagelist, length, faulted_page))" />

<action>
<!-- CDATA[
  if (object->declaredBehaviour == SEQUENTIAL) {
    vm_map_lock(map);
    plan_and_alloc_sequential_prefetch_pages(object, pagelist, length, faulted_page);
    vm_map_unlock(map);
  }
]-->]
</pointcut>
</action>
</advice>

<advice type="after">
<pointcut params="struct uio * io_info, int size, struct buf ** bpp">
<expression>
<!-- CDATA[
  if (object->behaviour == SEQUENTIAL) {
    vnode* vp = object->handle;
    struct uio* io_info =
      io_prep(pagelist[faulted_page]->pindex, MAXBSIZE, curproc);
    int error = ffs_read(vp, io_info, MAXBSIZE, curproc->p_ucred);
    return cleanup_after_read(error, object, pagelist, length, faulted_page);
  } else
    proceed(object, pagelist, length, faulted_page);
]-->]
</pointcut>
</advice>
Listing 5.1: XML specification for a prefetching aspect for the AspectC transformer

```xml
# include ...

aspect sequential_mapped_file_prefetching {

pointcut fault_path (vm_map_t map):
cflow (execution (int vm_fault (map, vm_offset_t, vm_prot_t, int)));

pointcut ffs_read_path (struct vnode* vp, struct uio* io_info, int size, struct buff ** bpp):
cflow (execution (int ffs_read (vp, io_info, size, bpp)));

before (vm_map_t map, vm_object_t object, vm_page_t* pagelist, int length, int faulted_page):
execution (int vnode_pager_getpages (object, pagelist, length, faulted_page))
&& fault_path (map) {
  if (object->declared_behaviour == SEQUENTIAL) {
    vm_map_lock (map);
    plan_and_alloc_sequential_prefetch_pages (object, pagelist, length, faulted_page);
    vm_map_unlock (map);
  }
}
```
around (vm_object_t object , vm_page_t* pagelist , int length , int faulted_page) :
exection (int ffs_getpages (object , pagelist , length , faulted_page)) {
    if (object->behaviour == SEQUENTIAL) {
        struct vnode* vp = object->handle ;
        struct uio* io_info = io_prep (pagelist[faulted_page]->pindex ,
                                      MAXBSIZE , curproc);
        int error = ffs_read (vp , io_info , MAXBSIZE , curproc->p_ucred);
        return cleanup_after_read (error , object , pagelist , length ,
                                   faulted_page);
    } else
        proceed (object , pagelist , length , faulted_page);
}

after (struct uio* io_info , int size , struct buf** bpp) :
exection (int breadn (struct vnode* , daddr_t , int , daddr_t* ,
                    int* , int , struct ucred* , struct buf**))
    && fault_path (vm_map_t , vm_offset_t , vm_prot_t , int)
    && ffs_read_path (struct vnode*, io_info, size, bpp) {
        flip_buffer_pages_to_allocated_vm_pages ((char*)bpp->b_data ,
                                                size , io_info);
    }
}

Listing 5.2: Prefetching aspect generated in AspectC

This static aspect specifies an allocation of virtual pages for prefetched pages in
the first advice, and their subsequent de-allocation in the event it is not cost-effective
to retrieve them in the next three advice. This functionality is a refactoring of an
optimization commonly found within FreeBSD, but in this form it is an optimization
that can be added/removed easily at compile-time.

Rustic management tools exist for understanding behaviour associated with this
and other virtual memory optimizations. For example, many Unix operating systems
such as FreeBSD support system calls such as vmstat, shown in Figure 5.1. For ex-
ample, vmstat pauses a given number of seconds between each display, then reports
Prefetching is a common feature in operating system code. In SONAR, not only is its structure improved through modularization as an aspect, but in this form the concern is better coupled with the tools that monitor resources it is designed to optimize. In future work, we will consider the ability to tune this optimization further, according to application specific needs and across the kernel boundary.

5.2 Virtual Machine Navigation

GCSpy is a general purpose heap visualization framework that allows developers to perform dynamic analysis of memory consumption [48]. It is designed to visualize a wide variety of memory management systems, offering dynamic visualization of workloads for either garbage collected or manually controlled systems.

GCSpy is based on a client-server architecture where the system that is being visualized is the server and the visualization GUI is the client, communicating through standard sockets. A screenshot of the client is shown in Figure 5.3, highlighting used versus unused space in memory, and supplying an interface for the full suite of GCSpy.
Figure 5.3: This screenshot shows GCspy visualizing Sun’s Java HotSpot virtual machine running the SPECjvm98_213_javac benchmark.

functionality. The advantages of this client-server design are that a minimal amount of code is added to the system being visualized, as the client can be run on another machine to affect the operation of the server as little as possible, and the client can be connected to and disconnected from the server anytime. Only the server side of GCSpy needs to be customized for a particular system, and the client is a portable GUI, making it suitable as a management tool within SONAR.

Here we consider the application of the SONAR model for introducing a GCSpy aspect within the Jikes Research Virtual Machine (RVM) [2, 24]. The XML specification of a small portion of the GCSPY aspect for AspectJ is shown in Listing 5.3, and the corresponding aspect is shown in Listing 5.4. The management tool is unchanged from that shown in Figure 5.3. This code shows two simple advice, and highlights a key issue associated with the SONAR model. The issue highlighted here deals with effective refactoring for SONAR.

```xml
<xml version="1.0" encoding="UTF-8"/>
<sonar>
<system name="gcspy">
<aspect name="gcspy" class="org.mmtk.plan.GCSPY"
baseClass="org.mmtk.plan.GCSPYBase"
deployment-model="perJVM" language="AspectJ">
<import>org.mmtk.vm.gcspy.*</import>
```
<pointcut name="planBoot" expression="execution(*
    Plan.boot())" />
<advice name="planBoot" type="before" bind-to="planBoot">
    <action>
        <![CDATA[
            planBoot();
        ]]>
    </action>
</advice>

<pointcut name="planPostAlloc" params="VM_Address ref, 
    Object[] o, int bytes, boolean b, int allocator"
    throws="VMPragmaUninterruptible">
    <expression>
        <![CDATA[
            execution(* Plan.postAlloc(VM_Address, Object[], int, 
                boolean, int)) 
            && args(ref, o, bytes, b, allocator)
        ]]>
    </expression>
</pointcut>
<advice name="planPostAlloc" type="around" bind-to="planPostAlloc" autoProceed="false">
    <action>
        <![CDATA[
            if (! planPostAlloc(ref, o, bytes, b, allocator)) {
                proceed(ref, o, bytes, b, allocator);
            }
        ]]>
    </action>
</advice>
</aspect>
</system>
</sonar>

Listing 5.3: XML specification for GCSpy aspect for AspectJ transformer

package org.mmtk.plan;

import org.mmtk.vm.gcspy.*;
privileged aspect GCSpy extends GCSpyBase {
  before():
  execution(* Plan.boot()) {
    planBoot();
  }

  void around(VM_Address ref, Object[] o, int bytes, boolean b, int allocator) throws VM_PragmaUninterruptible:
  execution(* Plan.postAlloc(VM_Address, Object[], int, boolean, int))
  &\& args(ref, o, bytes, b, allocator) {
    if(! planPostAlloc(ref, o, bytes, b, allocator)) {
      proceed(ref, o, bytes, b, allocator);
    }
  }
}

Listing 5.4: GCSpy aspect generated in AspectJ

With respect to the refactoring issue, the incarnation of the GCSpy aspect that is appropriate for SONAR is different from the first refactoring of the GCSpy aspect presented originally in [24]. In SONAR, raw code in the XML template should be minimized, as it will only be exposed to a rudimentary XML editor and not full IDE support. Hence, the form of the aspect presented here leverages a helper class to include the bulk of the implementation, reducing the XML definition to the dependency that exists between the pointcut/method signatures. This way the core functionality of GCSpy can be still edited in the rich context of tool support present in the IDE, instead of a plain XML editor.

The GCSpy aspect demonstrates a simple application of the SONAR model applied at the VM level. In this example, the original implementation is coupled with a predefined monitoring tool. Coordinating this example with other memory management concerns controlled by SONAR in the operating system (such as the previous example), and other monitoring tools with feedback capabilities, provides a viable means for a system-wide approach to memory management.
5.3 Application Level Navigation and Optimization

This section offers examples from two different sample applications. The first is based on a modified version of the J2EE sample banking application, *Duke’s Bank*, and makes use of SONAR’s transformer for AspectWerkz (Section 5.3.1). The second is based on a modified version of Spring.NET’s sample airline reservation system, *SpringAir*, and makes use of the Spring.NET AOP transformer (Section 5.3.2). The associated costs of the deployment of dynamic aspects such as these are further evaluated in Section 5.4.

5.3.1 Navigation of a Simple Banking Application

The following example, consisting of *RequestAspect* and *ProfileAspect*, demonstrates SONAR’s ability to navigate a sample system on a per-request basis. It is based on a modified version of the J2EE sample banking application, *Duke’s Bank*. Given that the scenarios considered in the example are request-centric, the modification to the code was to introduce on-demand per-request profiling built on top of Sonar’s supporting infrastructure (i.e., *Flows*, *Probes* and flow-aware logging). That is, a request can be associated with a flow object (supplied by the requesting client), and per-request profiling is performed only for the ones with a flow object. The following subsections explain in detail how these are achieved, respectively.

5.3.1.1 Flow-attached Request

First, we discuss how flow objects are attached to and extracted from requests. *Duke’s Bank* is a web application running on top of JBoss Application Server (with embedded Tomcat as web server). It listens and processes HTTP requests made from web clients (i.e., browsers). We decided to handle flow objects attachment and extraction at the HTTP connector layer in Tomcat, by using custom HTTP header or query string. The main advantage of this is that it does not require changes in application/framework specific payload data structure and, therefore, it can be reused in many HTTP-based applications. More specifically, a request can have a flow token (which is a flow object encoded in string form as illustrated in Section 4.1.3) attached as in a custom header or query string, named *flow-token*. 
As shown in Listing 5.5, RequestAspect is implemented using SONAR’s transformer for AspectWerkz, Flows and Probes APIs. Once being deployed, RequestAspect is woven with Tomcat’s org.apache.coyote.Adapter in order to intercept HTTP requests (represented as objects of type Request) before they are served by the application. It also registers a RequestMatcher, which is a criteria matcher bound to type Request, with ProbeManager for inspecting these requests. Upon receiving a request, RequestAspect reports it to the corresponding probe (which is also bound to type Request). As discussed in Section 4.2, during the reporting process, RequestMatcher is invoked to inspect the given request. Listing 5.6 illustrates how RequestMatcher searches for flow token in HTTP header or query string, and how it resumes a found flow via FlowManager. After the flow is resumed, a SessionHandle is returned to RequestAspect which then proceeds with the actual request processing by calling the JoinPoint.proceed() method. After the request processing completes, RequestAspect closes the obtained handle and thus ends the flow.

As discussed above, attaching a flow to a request is done by adding a flow token as either HTTP header or query string. An easy way to test this out is to add the custom query string to the requested URL manually, following the HTTP specification. For example, in Duke’s Bank, in order to display the accounts of a user, a client needs to access the following URL: http://localhost:8080/bank/accountList. To attach a flow to this request, the query string, ?flow-token=##:##:##:##, can be appended to the URL. As a result, the processing associated with this particular request sees an active flow resumed from the supplied token, and flow-aware activities (e.g., flow-aware logging in Section 4.3.3 and flow-aware profiling in the following section) are performed accordingly. The details regarding the flow token format are described in Section 4.1.3 and an example follows this section.

Moreover, since a flow token carries enough information for a flow to be resumed at remote sites, attaching it to requests allows Duke’s Bank to participate in distributed flows in a larger context. In other words, the request processing within Duke’s Bank can be correlated with the processing occurred in some other parts of the system. For example, suppose a client A is an application with Flows integrated. Prior to making a request to Duke’s Bank, A can branch a sub-flow from its current flow. The newly created sub-flow can then encoded and transmitted with the corresponding request, and it is resumed once the request reaches Duke’s Bank. As illustrated in Section
4.1.4, the resumed flow contains the parent stamp to trace back to the corresponding processing in $A$.

```java
import ...

public class RequestAspect extends ManagedAspectTemplate {

    RequestMatcher cm;

    public RequestAspect(AspectContext context) {
        super(context);
    }

    protected synchronized void afterDeploy() {
        // Register request matcher upon deployment
        cm = new RequestMatcher();
        getProbeManager().addCriteriaMatcher(cm);
    }

    protected synchronized void beforeUndeploy() {
        if (cm != null) {
            // De-register request matcher upon undeployment
            getProbeManager().removeCriteriaMatcher(cm);
        }
    }

    public Object service(Request req, Response res, JoinPoint thisJoinPoint) throws Throwable {
        Probe<Request> probe =
            getProbeManager().getOrCreateProbe(Request.class);

        SessionHandle<Request> handle = probe.report(this, req);
        try {
            getLogger().info("Logger=" + getLogger() + ", thread=" + Thread.currentThread().getName() + ", isFlowActive=" + getFlowManager().isFlowActive());
            return thisJoinPoint.proceed();
        } finally {
            handle.close();
        }
    }
```
Listing 5.5: Request intercepting aspect generated in AspectWerkz

```java
import ...

public class RequestMatcher implements CriteriaMatcher<Request> {
    static final String FLOW_TOKEN_KEY = "flow-token";

    public Class<Request> getDataType() {
        return Request.class;
    }

    public boolean match(Session<Request> session) {
        Request req = session.getData();

        // Search for flow token in HTTP headers and query parameters
        String flowToken = req.getHeader(FLOW_TOKEN_KEY);
        if (flowToken == null) {
            flowToken = req.getParameters().getParameter(FLOW_TOKEN_KEY);
        }

        // Resume flow if a valid flow token is found
        if (SonarUtils.isValidFlowToken(flowToken)) {
            SonarUtils.getFlowManager().resumeFlow(flowToken);
            return true;
        }

        return false;
    }
}
```

Listing 5.6: RequestMatcher

In this example, we have demonstrated how flows can be attached to requests made to Dukes Bank, and how RequestAspect inspects requests, extracts and resumes attached flows, using Flows and Probes APIs. These facilitate building request-centric navigation tasks/views, as demonstrated in the following section.
5.3.1.2 Flow-aware Profiling and Logging

The flow-attached request mechanism from the previous section provides the foundation for constructing on-demand per-request navigation tasks/views. Specifically, this section demonstrates two particular flow-aware tasks (profiling and logging) along with their artifacts. The runtime management of dynamic aspects is also illustrated.

ProfileAspect reflects a request-centric view of the system, recording key data points as requests are serviced. Similarly to RequestAspect, it is implemented using SONAR’s transformer for AspectWerkz and Flows API. It is not enabled (i.e., its deployment strategy is manual) when the system is started. However, the aspect itself is registered as a standard MBean (management bean) to the JBoss JMX server. The stakeholder can thus enable it through domain-independent deployment operations (i.e., deploy/undeploy), as illustrated in Section 3.3. Once being deployed, it is woven with the key components involved in serving requests to Duke’s Bank application: 1) HTTP connector in Tomcat (i.e., org.apache.coyote.Adapter); 2) SessionBeans (i.e., AccountControllerBean) for applying business logic; 3) EntityBeans (i.e., AccountBean) for data access; 4) JSP pages for response building (i.e., the generated _jsp classes). It then intercepts the calls to the selected methods (within the above components) during request processing.

As illustrated in Listing 5.7, upon intercepting a call, ProfileAspect first checks if an active flow presents in the current processing thread. If there is, it performs profiling for the underneath method and saves profiling results for later access, as in lines 42 through 50. If not, it immediately proceeds with the underneath method without any profiling, as in line 53. Since profiling is only performed for the requests with a flow attached, the flow id is used as the unique identifier (ReqID) for identifying these requests as well as their corresponding profiling results. As discussed in the previous section, flow tokens which include flow ids are attached to requests on the client side, before reaching Duke’s Bank. Therefore, it is clients’ responsibility to guarantee the uniqueness of flow ids.

```
1 import ...

2 public class ProfileAspect extends ManagedAspectTemplate implements ProfileAspectMBean {
    3     volatile boolean collectStackTraceEnabled;
```
volatile boolean collectTimeEnabled;

public ProfileAspect(AspectContext context) {
    super(context);
}

public boolean isCollectStackTraceEnabled() {
    return collectStackTraceEnabled;
}

public void setCollectStackTraceEnabled(boolean enabled) {
    collectStackTraceEnabled = enabled;
}

public boolean isCollectTimeEnabled() {
    return collectTimeEnabled;
}

public void setCollectTimeEnabled(boolean enabled) {
    collectTimeEnabled = enabled;
}

public ProfilingResults getProfilingResults(int id) {
    return ProfilingResultsStore.get(id);
}

public void setProfilingResultsBufferSize(int size) {
    ProfilingResultsStore.setSize(size);
}

public void clearProfilingResultsBuffer() {
    ProfilingResultsStore.clear();
}

public Object profile(JoinPoint thisJoinPoint) throws Throwable {
    if (getFlowManager().isFlowActive()) {
        StackTraceElement[] stackTrace =
            isCollectStackTraceEnabled() ?
            Thread.currentThread().getStackTrace() : null;
        }
    }
```java
long beginTime = isCollectTimeEnabled() ?
    System.currentTimeMillis() : -1;
try {
    return thisJoinPoint.proceed();
} finally {
    long endTime = (beginTime != -1) ?
        System.currentTimeMillis() : -1;
    ProfilingResultsStore.add(getFlowManager().getFlowId(),
        new ProfilingResult(AopUtils.getClassName(thisJoinPoint),
            AopUtils.getMethodName(thisJoinPoint),
            beginTime, endTime, stackTrace));
}
else {
    return thisJoinPoint.proceed();
}
```

Listing 5.7: Flow-aware profiling aspect generated in AspectWerkz

*ProfileAspect* exposes several key configurable optimization and navigation options and operations through JConsole, such as the ability to:

- configure profiling details (stack trace, timestamps, etc),
- manipulate result buffer operations (change the maximum size, clear the buffer).

All of the above options and operations are accessible through this aspect’s JMX interface (as defined in the *ProfileAspectMBean* interface). All data is stored on the server side and can be retrieved and viewed through JMX management tools.

Figure 5.4 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 adapter—the entry point of serving an HTTP request. Access to SessionBeans, EntityBeans and JSPs are traced to clearly show the processing time in each layer according to J2EE architecture.
Figure 5.4: The stack trace of serving a request that retrieves a list of customer accounts and their corresponding balances.
In addition to profiling, we modified Duke’s Bank’s logging to demonstrate SONAR’s flow-aware logging solution. The original implementation of Duke’s Bank logs key events via a class named Debug which in turn prints messages to System.err. The modified version of Debug uses SLF4J and logs messages at DEBUG level. SLF4J is setup to forward messages to the underneath Log4j and the global filtering threshold is set to INFO in Log4J’s configuration file, jboss-log4j.xml.

In Listing 5.8, line 1 is produced upon instantiating RequestAspect’s logger and the message states the SonarLogger wraps around a Log4jLoggerAdapter. Line 5 through 32 are produced while serving a request to http://localhost:8080/bank/accountList?flow-token=123:32. In the flow token, 123 is the flow id; 32 sets the PROMOTE flag (bit 2 as shown in Figure 4.2) but leaves the other settings to be as default; the parent stamp is left blank. The PROMOTE flag instructs SonarLogger to promote messages’ severity level if necessary while processing this particular request. More specifically, line 5 is produced by the INFO level log statement (line 29 in Listing 5.5) in RequestAspect. Line 6 through 32 are produced by various parts in the application via the modified Debug class. The ”-DEBUG-” in these messages indicates that they are promoted from DEBUG level. The ”[id=123, ld=1, lseq=1, t=75]” is added by SonarLogger to show flow states (i.e., flow id, local depth, local sequence, current thread id). By adding flow ids to messages, it is possible to identify and correlate them with their corresponding requests and with other artifacts, such as the profiling results collected by ProfileAspect.

```
16:17:24,187 INFO  [RequestAspect] Wrap logger
   [class=org.slf4j.impl.Log4jLoggerAdapter, threshold=INFO]
...
16:17:38,203 INFO  [RequestAspect] [id=123, ld=1, lseq=1, t=75]
   Logger=sonar.slf4j.SonarLogger@8a4f41,
   thread=http-127.0.0.1-8080-1, isFlowActive=true
16:17:38,203 INFO  [Debug]  -DEBUG-  [id=123, ld=1, lseq=1, t=75]
   /accountList
16:17:38,203 INFO  [Debug]  -DEBUG-  [id=123, ld=1, lseq=1, t=75]
   Forwarding to template.
16:17:38,234 INFO  [Debug]  -DEBUG-  [id=123, ld=1, lseq=1, t=75]
   CustomerControllerBean ejbCreate
16:17:38,296 INFO  [Debug]  -DEBUG-  [id=123, ld=1, lseq=1, t=75]
```
AccountControllerBean ejbCreate
10 16:17:38,328 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
TxControllerBean ejbCreate
11 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountControllerBean getAccountsOfCustomer
12 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
CustomerBean ejbLoad
13 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbLoad
14 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbLoad
15 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbLoad
16 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
CustomerBean ejbStore
17 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbStore
18 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbStore
19 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbStore
20 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbStore
21 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountControllerBean getAccountsOfCustomer
22 16:17:38,375 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
CustomerBean ejbLoad
23 16:17:38,390 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbLoad
24 16:17:38,390 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbLoad
25 16:17:38,390 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbLoad
26 16:17:38,390 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbLoad
27 16:17:38,390 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountControllerBean getAccountsOfCustomer
28 16:17:38,390 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
CustomerBean ejbStore
29 16:17:38,390 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
AccountBean ejbStore
30 16:17:38,390 INFO [Debug] -DEBUG- [id=123, ld=1, lseq=1, t=75]
Listing 5.8: Log from serving a flow-attached request

5.3.1.3 Summary

This case study illustrates how request-centric navigation tasks/views can be built with dynamic aspects within the SONAR model, using SONAR’s supporting infrastructure. Moreover, it demonstrates how JMX-based management tool provides a means of managing aspects’ deployment, tuning operations and retrieving data. Though the tools from the previous examples have been far more rudimentary, we envision this kind of management facility to be applied system-wide—across application, VM and operating system boundaries.

5.3.2 Cross-Platform Optimization

We chose SpringAir, a web application built upon Spring.NET framework, as our last case study. Specifically, this study demonstrates SONAR’s cross-platform support for optimization. The basic idea is to retrieve some cached data from remote systems and build a local copy in order to save the time spent on network communication. Optimizations that improve locality can dramatically impact the evolvability of many of today’s web-service based applications.

We developed an AirportInfoCacheAspect which provides caching for airport information. As shown in Listing 5.9, the airportInfoCacheAdvice is applied to the DefaultBookingAgent’s service methods with names matching the GetAirport.* pattern.

Spring.NET AOP is quite different from other AOP frameworks. Therefore, the XML specification schema is extended in order to address such differences. The most noticeable changes are made to the pointcut tag, since Spring.NET AOP does not define an AspectJ-like pointcut expression language. Furthermore, in Spring.NET
AOP, advice must implement one of the built-in advice interfaces (i.e., `IMethodInterceptor`). As a result, a class cannot define two advice of the same type. Therefore, as in line 12 and 13, the `class` attribute and `import` tag are added to the `advice` tag.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sonar>
  <system name="springAir">
    <aspect name="airportInfoCacheAspect" objectID="bookingAgent"
      deployment-model="perClass" language="Spring.NET">
      <pointcut name="airportInfo" type="RegexpMethod">
        <target-type>SpringAir.Service.DefaultBookingAgent</target-type>
        <target-method-pattern>GetAirport.*</target-method-pattern>
      </pointcut>
      <advice name="airportInfoCacheAdvice"
        class="SpringAir.Cache.AirportInfoCacheAdvice"
        type="around" bind-to="airportInfo" autoProceed="false">
        <import>SpringAir.Domain;Sonar.Cache</import>
        <action>
          <![CDATA[
            string airportInfoName = 
              invocation.Method.Name.Substring(3);
            object airportInfo = GlobalCache.get(airportInfoName);

            if(airportInfo != null)
              {
                return airportInfo;
              }

            airportInfo = invocation.Proceed();

            GlobalCache.insert(airportInfoName, airportInfo);

            return airportInfo;
          ]]>}
          </action>
        </advice>
    </aspect>
  </system>
</sonar>
```
Listing 5.9: XML specification of cache aspect for Spring.NET transformer

```csharp
using SpringAir.Domain;
using Sonar.Cache;

using AopAlliance.Intercept;
using Spring.Aop;

using System;

namespace SpringAir.Cache {
    public class AirportInfoCacheAdvice : IMethodInterceptor {
        public object Invoke(IMethodInvocation invocation) {
            string airportInfoName = invocation.Method.Name.Substring(3);

            object airportInfo = GlobalCache.get(airportInfoName);

            if (airportInfo != null) {
                return airportInfo;
            }

            airportInfo = invocation.Proceed();

            GlobalCache.insert(airportInfoName, airportInfo);

            return airportInfo;
        }
    }
}
```

Listing 5.10: Cache aspect generated in Spring.NET

```xml
<object id="airportInfoCacheAdvice" type="Spring.Aop.Support.RegexpMethodPointcutAdvisor">
    <property name="pattern" value="GetAirport.*" />
    <property name="advice">
        <object type="SpringAir.Cache.AirportInfoCacheAdvice" />
    </property>
</object>
```
Listing 5.11: Configuration XML generated in Spring.NET

Spring.NET AOP supports applying advice programmatically or declaratively using XML configuration. Listing 5.11 shows the generated XML segment.

Since there is no direct support for JMX in either Spring.NET or .NET framework, we decided to expose the cached information through a web service by using .NET’s web service support. In this way, the cached data is accessible by any programs with web services support. Our Java client uses the Java web service API to access the cached airport information in XML, and parses the data to build a local cache.

In addition to crossing application, VM, and operating system boundaries within a single node, this final example demonstrates how SONAR can apply across multiple systems. This opens the possibility of monitoring and tuning cross-platform concerns, and even further coordinating them with lower-level resource management conforming to the SONAR model.

5.4 Analysis: Costs and Benefits

Here we consider costs in terms of performance and memory, and benefits in terms of the strength of the overall programming model of SONAR. In terms of costs, with any approach to instrumentation there are overheads, but the overheads associated with the static aspects (Sections 5.1 and 5.2) are less substantial than those associated with dynamic AOP (Section 5.3). In Section 5.4.1 we focus on quantification of these costs associated with the dynamic aspects such as those introduced in Section 5.3.
Table 5.1: Impact on target class size. Both classes have a single around advice woven to a single method, respectively, using AspectWerkz 2.0.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (bytes)</td>
<td>7,738</td>
<td>25,521</td>
</tr>
<tr>
<td>Post weaving (bytes)</td>
<td>8,719</td>
<td>26,327</td>
</tr>
<tr>
<td>Increase (%)</td>
<td>12.68</td>
<td>3.16</td>
</tr>
</tbody>
</table>

With respect to benefits, Section 5.4.2 highlights the way the SONAR model to be useful, and the extensibility of this approach within other contexts.

### 5.4.1 Costs: Performance and Memory Utilization

All tests reported in this section were conducted on a Pentium 4 2.4GHz, 512MB machine, running Windows XP SP1, JDK 1.5.0 (1.5.0_02).

Since SONAR is designed not only for development systems but also for production systems, performance is crucial for its applicability. Moreover, system navigation tasks usually require introspection of system state, but the process of navigation might adversely impact system state and behaviour [27]. As a result, the performance impact of SONAR should be as minimal as possible. In this section we consider costs associated with SONAR. In order to get an impression of the worst case, we consider the most heavyweight approach supported by the prototype—the dynamic bytecode instrumentation provided by AspectWerkz.

Currently, no JVM supports schema redefinition of any loaded classes. That is, changes (such as add, remove or rename fields or methods, change of method signatures or inheritance) are not allowed at runtime\(^1\) [8]. However, dynamic deployment/undeployment of aspects in AspectWerkz requires schema changes to target classes.

To get around this restriction, AspectWerkz uses a preparation mechanism to enable target classes for later deployment/undeployment at runtime. A special construct

\(^1\)As mentioned in the Java API instrumentation section, this restriction might be annulled in the future.
called deployment scope is used to specify the join points to be prepared by adding a call to a public static final method that redirects to the target join point. The added indirection introduces overhead; however, such indirection can be inlined by most modern JVMs [8]. As a result, the impact of the added hook on runtime performance is negligible when aspects are not woven into or removed from target classes. Even for woven aspects, the time needed to instrument an advised method invocation has been optimized in the AspectWerkz 2 implementation [7].

Upon weaving or preparation, hooks are inserted into target classes. Consequently, the size of target classes is increased. Table 5.1 shows the impact of using AspectWerkz on target class size. The added size to each target class file is around 1,000 bytes. This does not include the size of aspect classes themselves since only the hook code is inserted into target classes. For medium and large classes, this is still relatively small. Table 5.1 also shows that as the original target class size increases from 7,738 bytes to 25,521 bytes, the impact decreases from 12.68% to 3.16%.

As shown in Figure 5.6, the JBoss Application Server’s startup time is increased by about three times. The performance is significantly degraded since it is running under AspectWerkz’ online mode—aspects are woven into target classes when they are loaded into the JVM. Furthermore, as shown in Figure 5.6, the memory footprint is also increased by about 28 megabytes (31.30 percent).
Figure 5.6: Impact on memory footprint. Sampled under the same setting as in Figure 5.5.
We should mention here that one limitation of SONAR is that it does not support distribution. As mentioned in future work however, we would like to explore the incorporation of a development such as AWED into the SONAR model. However, in SONAR’s current prototype, by using JMX, aspects deployed in a distributed fashion could ultimately be accessed and managed through the JMX Remote API.

5.4.2 Benefits: Programming Model and Unified Framework

The key benefit of SONAR is its model for a unified framework. We have demonstrated manifestations of this model in three different SONAR configurations: (1) using the AspectC transformer and low-level system interfaces for management (Section 4.1), (2) using the AspectJ transformer and a customized client-server management tool (Section 4.2), and (3) using the AspectWerkz transformer and standardized JMX management tools (Section 4.3). Each of these configurations promotes an integrated approach for optimization/navigation with corresponding management interfaces.

This establishes the efficacy of a relatively language-agnostic approach in the context of system-wide evolution. Of course, it is not completely language independent, but is extensible in that regard. Though the tradeoffs in terms of specific costs associated with dynamic approaches will need to be visited further in terms of impact within large systems, the fact that SONAR can be used to accurately identify performance bottlenecks at many levels in the system and thus be part of effective evolution strategies in a principled way may indeed outweigh performance compromises in some cases. We believe this model will continue to extend to other contexts, such as embedded and distributed systems. Future work is discussed further in the following chapter.
Chapter 6

Future Work and Conclusion

This work has demonstrated the ways in which SONAR’s combined use of AOP, XML, and management tools supports a more fluid and continuous approach to interactive software evolution. We believe this model enables safe and principled system-wide evolution. We have shown:

- How XML can be used to support a language-independent notation for the instrumentation of aspects.
- How these aspects can be used for optimization and navigation across the software stack.
- How services such as JMX provide management and visualization tools coupled with these aspects.

Our analysis reveals that the costs of SONAR in a dynamic heavyweight scenario are not to be overlooked. However, it is not unreasonable to assume that some of these costs may in fact be offset by the savings incurred when appropriate optimizations such as prefetching, caching, or operation reordering, can be effectively combined as a system evolves. We believe SONAR provides critical tool support for allowing software developers to better explore these tradeoffs.

6.1 Contribution

The contributions of this thesis are:
• A model for a unified framework: on top of this model, aspects targeted at different languages/frameworks can be built in a structured and uniform way. The common structure facilitates understanding and reuse. The transformation/code generation approach allows easy enhancement and extension. The management aspect makes work built-on top of SONAR more suitable for dynamic production environments.

• Infrastructure to support the model in distributed systems: Flows and Probes, available to both applications and aspects, facilitate distributed tracing on demand and/or on arbitrary application/analytical task specific criteria. Logs allows log messages, one of the important artifacts generated during program execution, to be selectively generated, collected and then correlated without changing existing logging statements in applications/aspects code.

• Prototype elements of the model assessed in a feasibility study: they demonstrate the possibility and feasibility of SONAR’s model and infrastructure in the following environments:
  – Prefetching in the FreeBSD Operating Systems, with simple monitoring using vmstat.
  – Garbage collection in the Jikes JVM, with simple monitoring using GCSpy.
  – Application level navigation and optimization on top of the JVM and .NET platforms, with platform-specific management tools (i.e., JConsole).

6.2 Future Work

The Flows facility in SONAR allows implementing end-to-end tracking on a per request basis. However, achieving this requires a thorough understanding in areas such as threading and remote invocation. Fortunately, some application frameworks abstract these concerns away from applications. For instance, J2EE has internal thread and object pools, and it handles the remote communication between EJBs. Therefore, future work would include creating tighter integration with such framework(s) so that flows can be propagated without being exposed to applications.

The current SONAR prototype includes transformers for AspectWerkz, AspectJ, Spring.NET AOP, and AspectC, respectively. As overviewed in the previous chap-
ter, future work would include exploring transformers for more AOP frameworks. Additionally, we believe it is necessary to focus on creating tighter aspect compositions for optimizations that are more explicitly coordinated between distinct layers in a system—from the operating system, virtual machine, middleware and application software—along critical execution paths. This work would necessarily include further investigation of how to unify SONAR with heavyweight, low-level tool kits in order to provide an efficient means of truly integrating tasks crossing all layers in the software stack.

Another important avenue for the future is to investigate a high-level aspect composition language specifically for understanding and explicitly coordinating multiple aspects for multi-level optimization. The problem of scale, in particular with dynamic aspects, is critical in a system such as SONAR that enables fine-grained modifications to system infrastructure software.

Though SONAR’s centralized repository allows developers to scan through all the dynamic aspects applied to the system at any given time, we believe we need to leverage semantic representation of system behaviour, and further target more automated comprehensive management of collections of aspects. For example, this semantic analysis could enable developers to more easily recognize when some combination of optimizations may actually interfere with each other and have conflicts, or to determine when the life-time of an optimization has essentially expired due to changes in the external environment.

Finally, the application of the SONAR model within embedded systems poses a new set of challenges in terms of costs and benefits. Changes in memory footprint or performance characteristics would not be acceptable for many embedded real-time systems. Give that the costs for most constructs in AspectC are minimal, we believe these systems can potentially benefit from SONAR’s model, as they sorely require more attention for increased tool support for application development.
Bibliography


