

AFRL-IF-RS-TR-2003-239
Final Technical Report
October 2003



SOFTWARE SURVEYOR

Object Services and Consulting, Inc.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K508

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-239 has been reviewed and is approved for publication.

APPROVED:

/s/

JAMES M. NAGY
Project Engineer

FOR THE DIRECTOR:

/s/

JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE Oct 03	3. REPORT TYPE AND DATES COVERED Final Jul 00 – Jun 03		
4. TITLE AND SUBTITLE SOFTWARE SURVEYOR		5. FUNDING NUMBERS C - F30602-00-C-0206 PE - 62702F PR - DASA TA - 00 WU - 07		
6. AUTHOR(S) David Wells and Paul Pazandak				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Object Services and Consulting, Inc. 6111 Baywood Avenue Baltimore, MD 21209-3803		8. PERFORMING ORGANIZATION REPORT NUMBER N/A		
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/ITB 3701 North Fairfax Drive 26 Electronic Pky Arlington, VA 22203-1714 Rome, NY 13441-4505		10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2003-239		
11. SUPPLEMENTARY NOTES AFRL Project Engineer: James M. Nagy, IFTB, 315-330-3173, nagyj@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 Words) This contractual effort researched and developed technology that enables systems to meet high assurance, high dependability, and high adaptability requirements. Object Services built gauges to collect, analyze and present information about how deployed instances of distributed software actually interact. The non-intrusive gauges illustrate the interaction patters, how far the effects of changes can propagate and whether an anticipated action is likely to be safe and identify subtle differences between environments that might be the source of puzzling misbehavior. The software gauges are suitable for use in profiling software applications constructed using JAVA.				
14. SUBJECT TERMS Software gauge, JAVA, DASADA, Adaptable software, configurable software, software surveyor, surveyor, software probes			15. NUMBER OF PAGES 106	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

TABLE OF CONTENTS

TABLE OF CONTENTS.....	i
1 Executive Summary	1
1.1 Administration	1
2 Problem Statement.....	2
2.1 Objective.....	2
3 Approach.....	3
4 Technical Results.....	4
4.1 Probes.....	4
4.2 Gauges.....	4
4.3 Event Infrastructure	5
4.4 Connecting to Architectural Models.....	5
5 Demonstrations & Tech Transfer.....	6
6 Publications and Presentations.....	7
7 Lessons Learned.....	8
7.1 Big Picture & Local Depth are Both Essential	8
7.2 Early Scenarios and Scenario Sharing	8
7.3 Using Research Quality Software	8
7.4 Shifting Objectives.....	9
8 Conclusions.....	10

Appendices.....	11
A. Project Overviews	12
(A-1) Taming Cyber Incognito, David Wells and Paul Pazandak, Working Conference on Complex and Dynamic Systems Architectures, December 2001, Brisbane, Australia.....	12
1 Introduction.....	13
2 The Form & Use of Models.....	13
3 Why Static Modeling is Insufficient.....	15
4 Perpetual Modeling.....	16
4.1 Information Sources.....	16
4.2 Information Profiling	17
5 Software Surveyor	18
5.1 Architecture.....	18
5.2 Current Implementation Status	19
6 An Example	20
7 Open Issues & Next steps	22
8 Bibliography	24
(A-2) Measures of Success, OBJS Report, October 2000	25
B. Technical Reports.....	30
(B-1) Survey of Existing Instrumentation Tools, Paul Pazandak, OBJS Report, October 2000.....	30
(B-2) Probe Architecture & Functionality, Paul Pazandak,December 2000.	34
(B-3) ProbeMeister: Distributed Runtime Software Instrumentation, Paul Pazandak and David Wells, 1 st Int'l Workshop on Unanticipated Software Evolution (USE), Spain, June 2002	42
1 Introduction.....	43
2 Overview of ProbeMeister.....	44
2.1 Related Work	45
3 ProbeMeister Architecture	46
3.1 Communication Manager.....	47
3.2 Probe Manager.....	47
3.3 Configuration Manager.....	48
3.4 User Interface.....	48
3.5 Other Interfaces.....	49

4 Probes.....	50
5 Issues.....	51
6 Plans.....	53
7 Acknowledgements.....	53
8 Related Work	53
C. Software Specification Sheets.....	55
(C-1) OBJS ProbeMeister	56
(C-2) OBJS Gauge Tool Set	57
(C-3) OBJS EnviroProbes.....	59
(C-4) OBJS XML2Java.....	60
D. User Manual.....	61
E. Demonstrations & Tech Transfer	65
(E-1) IntelliGauge TIE - Using Gauges Throughout the Software Lifecycle to Improve Internet Information Systems, IntelliGauge Project Team, October 2000.	65
(E-2) Software Surveyor: Dynamically Mapping Untamed Software Applications, OBJS Project Brochure, June 2001.....	76
(E-3) Software Surveyor: Dynamically Mapping Untamed Software Applications, OBJS Project Brochure, June 2002.....	80
(E-4) Demo Abstract, OBJS Report, June 2002	84
(E-5) White Paper – Dynamic Modeling and Analysis Tools for Cougaar	85
(E-6) White Paper – Ensuring the Robustness of Service Discovery Responses	90
Ensuring the Robustness of Service Discovery Responses	91
Executive Summary	91
Background.....	91
The Problem.....	92
The Root Causes	93
Nature of the proposed solution.....	94
Details, Architectural Fit & Technology to Build On.....	95
Competing approaches.....	97
Deliverables Schedule.....	98
(E-7) Report on Demonstrations of ProbeMeister Technology to the UltraLog Program.....	100

1 Executive Summary

The military of the future will increasingly rely upon "information superiority" to dominate the battlespace. Achieving information superiority will require software applications that are far larger, far more complex, and far more distributed than comparable applications in existence today. Such systems are highly dynamic, due to the physical movement of components, resource loss, changing missions, software upgrades, and change of organizational structure. Understanding the status and behavior of evolving systems is a daunting task that is not well addressed by existing tools or procedures.

This project developed and demonstrated a suite of runtime tools to enable this understanding. These tools (collectively called *Software Surveyor*), are designed to dynamically deduce the configuration and behavior of component-based software and to reflect that knowledge back into existing (primarily static) architectural models for analysis by other tools.

1.1 Administration

Participants:

- Dr. David Wells (PI)
- Dr. Paul Pazandak

Project Website: www.objs.com/DASADA

This Website contains contact information and links to most of our technical reports. This Final Report contains only papers containing the most up to date information on all topics. Additional papers showing intermediate steps we took to get where we are can be found on our project Website

2 Problem Statement

The military of the future will increasingly rely upon "information superiority" to dominate the battlespace. Achieving information superiority will require software applications that are far larger, far more complex, and far more distributed than comparable applications in existence today. Such systems are highly dynamic, due to the physical movement of components, resource loss, changing missions, software upgrades, and change of organizational structure. Knowing how such a system is currently configured, whether it is behaving as expected, what is causing any problems, and how it can be legitimately modified into a better configuration are essential to their reliable use. At present, there are few tools that address any of these issues in large systems outside of test harnesses during development. Valuable as these tools are during system development, they provide little help in the field.

2.1 Objective

Understanding the status and behavior of evolving systems is a daunting task that is not well addressed by existing tools or procedures. The objective of this project was to develop and demonstrate a suite of runtime tools to enable this understanding. These tools (primarily a collection of probes and gauges, collectively called *Software Surveyor*), are designed to dynamically deduce the configuration and behavior of component-based software and to reflect that knowledge back into existing (primarily static) architectural system models where it can be used by other tools to reason about correctness and evolvability.

3 Approach

Few modern software applications are static; their components, and the components' organization with respect to each other, evolve over time as the result of new requirements, bug fixes, performance improvements, feature enhancements, and changes in their environments as the systems with which they interact change. Architectural models of intended and actual organization and behavior are increasingly used to check correctness and as a basis for managing evolution. However, statically defined models are inherently incomplete because of outside influences such as actions by third-party software, movement of military units using parts of the application, or degradation of the operating environment.

Dynamically reconfigurable software is radically different from traditional static software. Component replacement is a common occurrence, and components are often generated on-the-fly from specifications of client requirements and service provider capabilities. Connections between components are changed frequently by a wide number of tool types. Even the types of the objects/data passed between components are malleable; programming language types are typically encoded in XML and reconverted for program use by translator tools that operate based on metadata stored in files associated with the data. In consequence, the developer of a software component may not know the identity of the other components with which the component will be interacting, the types or exact behavior of those components, their location, or even the types of the information actually being exchanged. Not only is it currently difficult or impossible to determine how connectivity has been decided, it is often the case that critical decisions are made without propagating the knowledge that a decision has even been made to the proper authorities. This combines to make runtime monitoring and debugging difficult, and introduces obvious security risks. Detailed knowledge of actual runtime configuration and behavior to complete existing static models is thus essential.

This requires the following, all addressed by *Software Surveyor*:

- A large, diverse, efficient, and flexible collection of probes to gather application and environmental data.
- An extensible collection of gauges to perform the analysis of raw data collected by probes.
- A communications framework to allow interoperation and incremental addition of new probes and gauges.
- A logical connection between the monitoring mechanisms and the architectural specifications to allow specifications to be used to drive monitoring and to reflect derived information back into the models.

Appendix A contains papers describing goals and approach in greater detail.

4 Technical Results

Technology produced as part of Software Surveyor is discussed below. Technology falls into the broad categories of probes, gauges, event infrastructure, and modeling.

Appendix B contains papers with additional details and Appendix C contains Specification Sheets for key software produced. Appendix D contains the User Manual for ProbeMeister. Additional information on how to run the various gauges is packaged with the release.

4.1 Probes

Probes gather low-level application and environmental information. Most probes are designed to work in specific environments, with the type(s) required for a given monitoring situation depending upon the data gathering requirements, the application's implementation language, source code access, operating environment, and security considerations. Thus, runtime profiling requires a diverse set of probes. We have developed a tool, *ProbeMeister*, to (automatically or under user control) insert probes into distributed Java applications at runtime by dynamic bytecode modification. *ProbeMeister* comes with an extensible library of useful probe types and has facilities for managing configurations of probes to allow sets of related probes to be managed together. *ProbeMeister*-inserted probes can be removed when no longer required to reduce overhead. With *ProbeMeister*, generic probes can be used to gather information or the user can define custom “probe plugs” in a few lines of Java to collect exactly the kinds of information required. Writing a probe plug requires only minimal Java programming skills. *ProbeMeister* can be used to instrument application-specific code or Java core classes such as File, URL, and I/O stream access which are generically useful for monitoring component interactions. *ProbeMeister* is compatible with all relevant Java standards and has in fact influenced the Java debug interface through interactions with Sun developers.

4.2 Gauges

Gauges combine monitored events from (potentially) multiple sources into meaningful models of application configuration, behavior, and resource consumption. Examples of gauges developed under this project include: a *Coalescer* that merges streams of separately collected event information and renders this information on a timeline; an *EventMonitor* that categorizes events by type and renders HTML- and XML-based displayable summaries with expandable detail; an *EventMerger* that unifies reports of the same event reported at differing levels of abstraction to provide a more complete and coherent picture; a *StackTracer* that converts streams of application events into a trace of program execution and emits an XML representation; *Historian* that archives execution traces and computes statistics of behavior; and a *Mapper* that provides a visualization of the time-based relationships between events of an application. Most gauges are Web-enabled so that their outputs can be delivered easily to wherever they are needed by human operators or other higher-level tools. Information determined by gauges can be

reflected back into architectural specifications using facilities provided by the DASADA gauge infrastructure so that models can be kept up to date.

4.3 Event Infrastructure

In conjunction with other DASADA projects, we developed and used common event and dissemination mechanisms to allow probes and gauges from a variety of sources to interoperate. These were refined as more probes and gauges were developed and the requirements become more clear. *Software Surveyor* utilizes these mechanisms. *Software Surveyor* uses the Siena event distribution mechanism (contact Alex Wolf at U-Colorado for the latest version of Siena). We do not guarantee that *Software Surveyor* will remain compatible with future versions of Siena.

4.4 Connecting to Architectural Models

Architectural models are used in two ways by *Software Surveyor*. First, architectural models of probes, gauges, and the application can be used to drive gauge and probe placement. A user (or a program) can specify at the model level that a particular property of the model is to be monitored. This drives a process of model combination that results in directives to place particular probes and gauges into particular places in the application and its environment. This allows users with familiarity with the application's overall behavior and structure but without knowledge of implementation details to initiate monitoring as needed to solve problems in the field. A second use of architectural models by *Software Surveyor* is that information deduced by gauges can be reflected back into the architectural model of the application to allow third-party tools (some of which are developed by other DASADA projects) to determine if the system is operating within specified bounds and initiate corrective actions if this is not the case. Model feedback was demonstrated as a narrow path concept demonstration. It is not a mature tool at this time.

5 Demonstrations & Tech Transfer

Software Surveyor tools have been applied in two annual DASADA demonstrations to an intelligence gathering and analysis tool currently in use in the US Navy Pacific Command (PACOM). This provides a technology transition path and, equally importantly, forced us to develop probes and gauges that can be applied even when the application is incompletely specified (a key goal) and brought to the forefront a number of challenging issues related to ambiguity in the precision of the monitored events (a simple example is clock skew in distributed applications). Software Surveyor is now mature enough that it can be evaluated in the context of a number of demanding applications as part of a planned series of large-scale experiments (described below). Appendix E contains more details.

As part of an expected DASADA Phase II effort to inject DASADA technology (including *Software Surveyor*) into a number of DoD applications and to measure the utility of that technology, we supported the efforts of several DoD organizations in writing white papers describing potential Phase II activities.

- PACOM – experiment with the use of DASADA tools in the context of a loosely coupled Internet-based intelligence gathering tool called GeoWorlds that is currently in use in PACOM
- TACOM – experiment with the use of DASADA tools in the context of the Future Combat System (FCS) as part of managing runtime reconfiguration of on-board software in response to battle damage and evolving mission needs
- AFRL – experiment with the use of DASADA technology in monitoring and maintaining the functionality of a Master Caution Panel being developed to manage the software in AF Air Operations Centers
- Common Infrastructure – continued development of a common event dissemination and control infrastructure to be shared across Phase II projects to reduce costs and enable interoperation

Three of these efforts were selected for Phase II follow-on, but funding for Phase II was withdrawn. As a result, this work will not be continued at this time. All white papers have been provided to the government, but are not included in this report since they are the property of the DoD Service organizations leading the efforts.

Three potential uses of ProbeMeister in the DASADA UltraLog Program were demonstrated to participants in the UltraLog Program. These uses were: for Red Teaming, distributed stress injection and data collection, and distributed debugging of a messaging subsystem. In each case, ProbeMeister was applied to a small, but live Cougaar society. Several groups in UltraLog are evaluating the use of ProbeMeister for these purposes.

ProbeMeister has been downloaded by 5 R&D and assessment groups. It is currently being used for instrumenting an implementation of the Joint Battlespace Infosphere at the Air Force Research Laboratory (AFRL).

6 Publications and Presentations

The following papers (see also Appendix) appeared in refereed conferences proceedings:

- Mapping Cyber Incognito, David Wells and Paul Pazandak, Working Conference on Complex and Dynamic Systems Architectures, December 2001, Brisbane, Australia.
- ProbeMeister: Distributed Runtime Software Instrumentation, Paul Pazandak and David Wells, 1st Int'l Workshop on Unanticipated Software Evolution (USE), Spain, June 2002.

The project results have been presented at the following workshops and meetings.

Copies of the overheads used appear on our Website and were distributed at the meetings.

- DASADA Kickoff Meeting – Santa Fe, NM, Sept 2000.
- DASADA Winter PI Meeting - Monterrey, CA, Jan 31 – Feb 2, 2001
- DASADA Demo Days & PI Meeting – Baltimore, MD, July 2001.
- Working Conference on Complex and Dynamic Systems Architectures, Brisbane, Australia, December 12-14, 2001.
- DASADA PI Meeting – Brisbane, AU, December, 2001
- DASADA 2002 Demo Days & PI Meeting – Baltimore, MD, July 2000

7 Lessons Learned

7.1 Big Picture & Local Depth are Both Essential

Monitoring, mapping, and controlling evolving software requires many different kinds of tools. As such, it is necessary to take a “big picture” view of the problem to avoid designing a solution to a specific part of the problem that is totally incompatible with anything else and is thus not useful. At the same time, to be useful, the tools have to be sufficiently robust that they can address sizable problems and be used by others who are not willing to “baby sit” temperamental tools that only work under limited circumstances.

In this regard, we believe that we accomplished our goals quite well. We were able to demonstrate in a narrow path, a complete cycle using application models to aid in probe placement, actual probe insertion, collecting monitored events over a standard bus, coalescing and analyzing the monitored events through a small collection of gauges, visualizing the gauged information in a user friendly way using Web browsers, and feeding the information back into the original models for future use. At the same time, our probe tool (ProbeMeister) is quite mature, is based on Java standards, has been evaluated by Sun Java engineers as being compliant, and in fact influenced part of the Sun debugging interface. Many of the gauges are also generally useful on their own.

7.2 Early Scenarios and Scenario Sharing

Our coding efforts were most successful after we were able to scope the development through a semi-realistic scenario. Software Surveyor as designed encompasses a wide range of capabilities; a far larger number than can reasonably be implemented in a project of this size. The IntelliGauge TIE scenario centered around GeoWorlds allowed us to focus on only those techniques required for that scenario, while at the same time maintaining consistency with other projects. Being able to share a test application developed by other program participants reduced the work factor, ensured that we were not solving artificial problems of our own creation, and having access to the developers allowed us to understand their real diagnostic and modeling “care-about’s”. The fact that the test application was being used externally in DoD provided additional realism and a tech transfer opportunity.

7.3 Using Research Quality Software

As usual when attempting to use other research quality software, the developmental maturity of that software was an issue. The quality of the systems we used varied. While this issue really has no resolution (R&D use of other R&D software is inevitable and desirable), a useful thing to keep in mind is that it is desirable to restrict the number of such evolving software whenever possible and to use more stable (even if less interesting) software when the “bang for the buck” of using the most advanced software is limited in a particular context.

7.4 Shifting Objectives

DASADA was plagued by uncertainty about the follow-on Phase II experiments, which ultimately were not funded. In any project, it is very helpful to have a clear idea of duration, as this allows better scoping of the design and development activities. In particular, if it is known that a program is terminal, then work is done to reduce the number of development activities and to achieve higher quality results with the ones that remain. On the other hand, if the project will continue, and especially if the continuation is part of a larger experiment, the proper thing to do is to seek the tool coverage necessary for the experiments, even if this means that the individual tools are less mature, since there will always be time to improve them during the course of the experiments. This was not possible in DASADA.

8 Conclusions

We feel that the project was quite successful, although we did not get as far as we had hoped. We are quite satisfied with our overall vision of a closed model-monitor-analyze-model loop for addressing the complexities of evolving, underspecified systems and the fact that we could demonstrate this complete loop in a real example application. We are also very satisfied with the probe software we developed for dynamic monitoring of Java programs. ProbeMeister has proven itself to be quite useable and has been employed in a variety of ways we had never previously considered, particularly red teaming and stress injection for testing.

We are actively trying to migrate ProbeMeister into a key role in debugging and assessing the Cougaar agent architecture as part of the DARPA UltraLog program. This will be an excellent tech transfer vehicle, because not only is Cougaar the underpinnings for a future military logistics system, it is also expected to play a key role in the Future Combat Systems software.

We are pursuing additional funding to continue this work, through our UltraLog connection and through SBIRs. We are also exploring the commercial potential of ProbeMeister. Continuation of this work is critical to the success of componentware and is not being addressed elsewhere.

Appendices

(See also www.objs.com/DASADA for additional documents)

A. Project Overviews

**(A-1) Taming Cyber Incognito, David Wells and Paul Pazandak,
Working Conference on Complex and Dynamic Systems Architectures,
December 2001, Brisbane, Australia.**

Abstract: Static models derived from specifications are inherently inadequate for capturing the reality of dynamic, reconfigurable software. Instead, continually updated models that combine static and dynamic information about software requirements, architectural patterns, components, connectivity, actions, and resource utilization are necessary. The Software Surveyor is an extensible toolkit for collecting, disseminating, and analyzing such dynamic information. The architecture and current status of Software Surveyor are presented, and the system's use is illustrated through an example application in the information fusion domain.

Taming Cyber Incognito

Tools for Surveying Dynamic/Reconfigurable Software Landscapes

David L. Wells and Paul Pazandak
Object Services and Consulting, Inc.
{wells, pazandak}@objs.com

Abstract

Static models derived from specifications are inherently inadequate for capturing the reality of dynamic, reconfigurable software. Instead, continually updated models that combine static and dynamic information about software requirements, architectural patterns, components, connectivity, actions, and resource utilization are necessary. The Software Surveyor is an extensible toolkit for collecting, disseminating, and analyzing such dynamic information. The architecture and current status of Software Surveyor are presented, and the system's use is illustrated through an example application in the information fusion domain.

1. Introduction

The power and flexibility of modern software makes the software landscape increasingly a *cyber incognita*, analogous to the *terra incognita* (unknown territory) that baffled explorers, frightened merchants and impeded progress hundreds of years ago. *Cyber incognita*'s equivalent of maps, surveying instruments, and marked trails are design specifications, monitoring and diagnostic tools, and descriptions of applications' normative behavior; all are as inadequate in *cyber incognita* today as their equivalents were in *terra incognita* 200 years ago. Design specifications are incomplete, inaccurate, or inconsistent; software probes cannot observe all significant events; techniques to correlate independently recorded observations are limited; and descriptions of normative behavior are often (especially in Web-based, agent, or survivable systems) described as "best effort" with no concrete notion of what that means. Further, the dynamic nature of many modern applications means that they are continually reorganizing themselves in response to changed user demands or resource availability; imagine Lewis and Clark having to deal with rivers and mountains

that changed position every few hours. *Hic sunt dracones* – here are dragons.

A multi-faceted approach to remedying this situation is needed, including: 1) formal, static specifications of required/expected behavior, and 2) dynamic, runtime tools to flesh out the static specifications and to verify that the application is adhering to specifications. Software Surveyor [1], a framework and an extensible set of probes and gauges to dynamically deduce the connectivity and behavior of evolving, under-specified software applications being developed as part of the DARPA DASADA Program [2], provides many of the required dynamic capabilities and is compatible with the coming generation of modeling tools.

Section 2 of this paper discusses the form and uses of application models. Section 3 argues that static techniques are inherently insufficient for modeling modern software, while Section 4 discusses how static and dynamic analysis together can provide better models and discusses the kinds of information that can be obtained from various points in the software lifecycle. Section 5 presents Software Surveyor; a suite of architecturally related tools to collect, disseminate, and analyze information collected by runtime application monitoring. Section 6 presents an extended example of the use of Software Surveyor as applied to a loosely coupled Internet information analysis application. Section 7 identifies future work.

2. The Form & Use of Models

A *model* is an abstraction of a real system that captures the essential elements, organization, and activities of that system. Models can define "families of systems" or can define a specific instantiation of a system. The xArch system [3] based on the Acme architecture definition language (ADL) [4] makes this distinction explicit and uses the same modeling language to define models at each level.

Models can be used to constrain system organization or behavior and provide a basis for reasoning about, simulating, or validating behavior. For example:

How are the components connected? Do all connections meet specifications? Are all required bindings satisfied? Are unexpected components present? Do seemingly valid bindings produce expected behavior?

How, when, and why were the connections made? Who/what is responsible for an incorrect binding? Can a specific binding be changed? Is some binding tool (e.g., a Trader) using a bad selection policy? Are binding decisions made consistently?

What is the physical organization? Where are the components physically located? Where are choke points? Are untrusted machines being used? Are components vulnerable to physical assault/failure?

How does the current configuration compare to other configurations? Is the configuration consistent with the specification? How does a faulty configuration differ from a known good configuration? What differences exist between a currently faulty configuration and a previously working configuration?

Has a configuration changed? What changed? What process changed it? Was a change authorized? Did an authorized change actually happen? Does a change indicate a possible intrusion or failure?

Are there unused or unexpected components? This provides an opportunity for pruning the configuration to reduce footprint, to identify potential viruses and Trojan Horses, and to simplify the evolution process by not evolving unused modules.

What are the activity patterns? Are QoS constraints met and are there hot spots? Are there patterns of connection quiescence? This provides input to resource allocation & optimization tools and helps to identify “suspicious” activity. It can also be used to allow unused connections to drop safely and identify “windows of opportunity” for evolution.

How are the components interacting? What functions are invoked on the various connections? This is useful for ensuring compliance with specifications, security monitoring, and general diagnostics.

Are there unused functions/methods of libraries/components? This can allow more specific library loading to reduce code footprint and simplify evolution by only upgrading functionality actually in use.

A model is inherently an approximation of the system being modeled. The approximation occurs because the model suppresses (unnecessary) details or because the model is incorrect in some respect(s). This requires an understanding of:

- What constitutes an “essential element, organization, and activity of the system”.
- How those items can be determined.

In this work, we consider *componentware*; software that is assembled by composing immutable, preexisting parts, possibly using “glue” software to facilitate the composition. The model of a componentware system is an annotated graph, whose nodes correspond to the immutable components and whose edges correspond to bindings (connections, whether actively used or not) between components. This logical organization constitutes the *topology* of the model. There is also a *geometry* corresponding to the physical computing, storage, and communications resources on which the components execute. Both may change over time, and the geometry may change while the topology remains constant (e.g., a process is relocated) [5].

Thus, the following are the salient constructs of a configuration:

- The immutable components
- The logical connections between components (who calls whom & protocols used)
- The resources on which the components execute (hardware & software environments)
- The connection medium (physical paths)

Applications can be profiled at many levels of abstraction. Since the point of modeling is to support some set(s) of users, it is appropriate to choose level(s) of abstraction that are meaningful to them. This means that the granularity of the model should be such that the modeled components are familiar to the users and that use of the model can point to practical remedial actions (e.g., restart a service, use alternate communications, choose a service alternative, do without a non-responsive service) that can be taken given the skills and tools available to the users.

Matching the level of abstraction to the actions that can be taken is particularly important, since if the proper tools are lacking to make a change, a portion of the application is immutable *to that class of users* and therefore should be considered as a component *from that point of view* regardless of how complex it might actually be. For instance, modifying the implementation or installation of a remote service might not be allowed, but switching to an alternate service might be. In this case, the knowledge that the remote service has failed is sufficient; details about how and why it failed are of no use and only create mental clutter. In Section 6 we will see how these concepts are applied in practice.

Note however, that a model is actually a hypergraph. An immutable component in one abstraction may in fact expand into a graph of smaller components in a finer-grained view or a view from a different perspective. This is good, because it again allows viewing a configuration at

a useful level of detail for the task at hand. An implication of this is that *Software Surveyor* must coordinate its collection, analysis, and presentation activities based on the desired view(s). This notion is called *focus*, and is discussed further in Section 7.

3. Why Static Modeling is Insufficient

It has always been impossible to completely characterize everything important about large, distributed applications, but with yesterday's relatively static applications it was possible to specify much of the relevant information as part of the design, implementation, or deployment processes and then to test in a constrained operating environment to ensure that the desired behavior was (more or less) achieved prior to actual use. Often, this was not particularly well done (especially when relying on multi-source components from vendors with varying quality controls and documentation standards), but at least there was a hope that with better tools, methodology, or training, it could be accomplished.

However, this is a vain hope with modern, loosely coupled software that is often constructed from a mix of custom and preexisting components originating from a variety of sources. Individual components can (and frequently do) evolve independently due to new requirements, bug fixes, performance improvements, and feature enhancements. Field upgrades of deployed code (e.g., by providing new libraries) can unwittingly cause problems for other programs that had previously been performing correctly (e.g., a DLL is upgraded to support application A, but causes problems for application B which also uses it). Lack of complete dependency information makes it impossible to know what might be affected by the upgrade or even to know that a subsequent malfunction might be related to a particular change. New components may be introduced, including components that are generated on-the-fly from specifications of client requirements and service provider capabilities and are never seen by a human or subjected to normal testing. Even the types of the objects/data passed between components are malleable; programming language types are typically encoded in XML and later reconstituted for program use by translator tools that operate based on metadata stored in files associated with the data.

Flexible architectures with loose inter-module coupling has many well known advantages, but in consequence, the developer of a component or application may not know the identity of all components, the types or exact behavior of

those components, their location, or even the types of the information actually being exchanged. This makes it very difficult to predict the effect of proposed changes, to determine why something does not work properly, or even to figure out why something works well in one environment but does not work in a seemingly comparable environment.

Further, much of the new software is designed to make many of its configuration decisions on the fly, depending on its environment. Frequently, these decisions are outside the direct control of the application developer. This includes mobile code that binds to local instances of services, CORBA services that are bound to existing servers by a Trader, and survivable systems that reconfigure to use remaining resources after attack or failure. Not only is it currently difficult or impossible to determine how connectivity has been decided, it is often the case that critical decisions are made without propagating the knowledge that a decision even has been made back to the proper authorities.

Finally, the operating environment is frequently too complex to replicate for testing purposes (imagine replicating the Internet to test software that filters and streams time-critical data over an open network).

The key observation is that it is becoming increasingly difficult to know in advance how components *actually* use each other due to greater system complexity combined with more dynamic configuration choices. Design specifications, architecture descriptions and formal methods, configuration information produced during the process of instantiating and deploying code, and tools that profile systems under development all provide valuable information, but even collectively they still leave notable gaps in the community's ability to gather "ground truth" about the real-world behavior of distributed, component-based systems. A brief examination of these sources of information shows why this is so.

Formal specifications are good for describing desired behavior in a way that supports reasoning about system properties; however, implementation details are difficult to capture this way and formal specifications for externally developed components are hard to come by. Since few systems work in isolation from all external components (including operating systems and communications software), formal specifications are necessarily incomplete.

Software construction tools (compilers, linkers, configuration management, etc.) that instantiate and manage software generate a large amount of information that is generally complete and accurate when produced. However, this kind of information suffers from one serious shortcoming: it may accurately reflect the connections that existed when the software was first created, but there is no feedback process to ensure that it remains accurate as the system evolves, particularly if the changes were caused by

a tool other than the one that produced the initial information. Further, such tools generally only identify the static interconnectedness graph of an application and not how those interconnections can be used.

Profiling tools found in software development environments do capture some of the dynamic behavior of systems. However, they have serious coverage gaps when considered in the context of component-based systems where key components are frequently outside the domain of the monitoring tools (wrong language, different platform, remote) and hence cannot be profiled. Even with integrated development environments that support multiple environments and distributed debugging, the problem remains that the tools are intended for use during the development phase rather than during the entire lifecycle of the system and as a result are too complex and resource intensive for everyday use with deployed systems.

Real, running component-based systems thus have behavior that cannot be adequately described without directly observing the behavior of the system “in the wild”. A major thrust of the *Software Surveyor* project is to construct living, constantly updated models of dynamic, under-specified applications by combining static information about the modeled system with information about binding decisions, component execution and interactions, and resource use collected *during runtime*. *Software Surveyor* fills a void left by more traditional tools that are employed *prior to program use*.

4. Perpetual Modeling

Information about application structure and behavior can be obtained from several sources, including *design artifacts*, *application artifacts*, *runtime monitoring*, and *historical information* about prior executions. However, no single source can provide all the information necessary to completely profile an application, so it is necessary to extract or collect information from all of these sources. Collectively, they:

- Identify the *kinds* of components and interactions that are important enough to profile,
- Provide a *conceptual framework* in which collected information can be organized,
- Tell *where to look* to collect the necessary information to allow a profile to be constructed,
- Provide *expectations* to which observed organization and behavior can be compared.

4.1 Information Sources

Design Artifacts: Design artifacts are descriptions of *intended* configurations and behavior. Static specifications limit undesirable behavior and mandate certain desirable behaviors. Static design specifications cannot deal adequately with the following kinds of dynamic behavior without unduly restricting the benefits of the dynamism: dynamic binding decisions by third-party binders, dynamic addition, deletion, or movement of independent data sources, changes to the schema of independent data sources and components, transient behavior. Static specifications cannot be arbitrarily fine grained and cannot generally anticipate all environments and conditions in which the software may be expected to operate. Further, enforcement of many types of design constraints, e.g., quality of service, requires runtime monitoring.

Application Artifacts: There are two kinds of application artifacts: information (such as configuration and source code files) that are below the level of design and are used to further reify the application’s configuration, and information that the application produces that is generally available without using probes. Both may require interpretation to be useful, but are easily captured. In addition to providing concrete information, they also indicate which events to look for; i.e., where to place probes. Generally, application artifacts do not provide sufficient insight into how the information was produced (the job of design artifacts and runtime monitoring) or what its purpose is (the job of design artifacts). Also, failures are particularly hard to analyze using only application artifacts. Finally, because separate executions are often totally independent, it is hard to detect anomalies from one execution to another.

Runtime Monitoring: Runtime monitoring of an application and its environment can add details that are left unspecified by design specifications and can identify the specific elements filling “roles” defined by the design. The specific elements might not have existed when the design specification was made and/or might be selected by third party software outside the control of the application. Finally, runtime monitoring is necessary to ensure that design constraints are being met. Runtime monitoring cannot tell *why* a particular event occurred, merely that it has. Interpretation must be with respect to design and application artifacts.

Historical Record: The historical record is the time-series behavior observed through runtime monitoring of multiple executions of the application. This can be used to informally determine expectations of behavior in any of the dimensions that can be monitored, and can serve as a basis for detecting anomalous behavior. In a sense, the historical record forms a piece of the design specification:

“it should work in a certain way because that’s the way it always has worked”.

4.2 Information Profiling

Profiling requires a diverse set of probes to collect information from the sources described above. Thus, a variety of probe types are needed to profile even reasonably complex applications. Reasons for using different kinds of probes include:

- Probes are designed to work in specific environments. The types of probes that will be required for a given deployment will depend upon the data gathering requirements, the application's implementation language, source code access, and operating environment.
- Probes have different information capture capability. Even if environmentally compatible, and monitoring the same event, different probes may be able to capture different information about the event. For example, when a process spawns a child process, a new Windows task is created. An application-specific or language-specific probe could capture the arguments used to start the new task, but information like process-id and memory utilization that could be used to externally monitor the task are better captured using either environmental probes or probes that monitor O/S events like process creation.
- Security and ownership concerns may mandate that only certain kinds of probes are allowed to be placed and that only certain insertion points are possible. This indicates another reason that choices in probe technology have value.

The following artifacts are of interest:

Component Types: The types of components that are significant to the intended users of the gauge outputs. The definition of the components of interest is dictated by the level of abstraction at which a particular class of users understands and manipulates the application. See [5] for a further discussion of how components of interest are determined. Once the interesting classes of components are identified, the classes of connections that must be profiled become obvious.

Architectural Patterns: Patterns (in the Gamma [6] sense) defining how components of interest can be connected together. Examples are trees, object buses, server farms, object factories, etc. An application may employ many patterns. The key point is that patterns define the way in which components will interact in the application, not which specific components fill the various roles in the patterns. Architectural patterns provide a framework within which components and connections may be interpreted.

Static Connectivity: Mandated connectivity between specific components of interest. This is more precise than an architectural pattern, since it specifies more detail and identifies particular components filling the various roles.

Interaction Protocols: The protocols by which components of interest are allowed to interact. Examples are HTTP, CORBA, Java RMI, SOAP, email, etc. Since component interactions are a prime place to insert probes, protocol documentation can tell where and how those probes can be inserted.

Initial Configuration: When applications are deployed, they have some initial configuration defined by mechanisms such as configuration files, registries, or the like. Some defaulting may be used to complete a partial specification (e.g., localhost is mapped to an IP address).

Information Output: Most applications produce some output in an easily accessible form such as displays, file or database writes, or the use of StdOut and StdError. Such output can represent the primary results of the application or can be diagnostic. In general it is easily captured without inserting probes directly into the application since it is intended to be exposed; however, it frequently requires parsing to interpret its meaning.

Dynamic Components: It is necessary to know the components actually in use and desirable to know the set of components available for use. Both can change over time. The available components may be explicitly expressed in some kind of registry, may be a set identified on-the-fly by a binding mechanism such as a Trader, or be generated on-the-fly from specifications (e.g., glueware) or from data accessed by the application. In addition to knowing which components are available and in use, it is desirable to know why (i.e., by what mechanism) they were selected.

Dynamic Bindings: Component connectivity must be tracked over time. This includes the components connected, the roles they play in a connection, and how/why the connection was established (e.g., the binding agent and the binding arguments). Dynamic bindings must be checked for adherence to the architectural patterns defined for the application.

Messaging Activity: This refers to all kinds of messaging between the components of interest. The types of these communications are defined by the interaction protocols identified elsewhere. It includes the initial

message, response, and any exceptions thrown. Note that exceptions need not be returned to the original caller, as the recipient will be defined by the interaction protocol.

Environment: Required (specified) and actual (sensed) environmental properties such as operating system, CPU speed, memory, disk, bandwidth, other required software should be captured and compared to determine if the requirements are met.

Extra-application resource utilization. How much of various resources are being used by other applications and are therefore unavailable to the application of interest? This is obviously a dynamic issue, in that the other co-resident applications will change from execution to execution. This is of interest since it may predict ability of the application to meet QoS requirements, user expectations (or may make it possible to warn the user that the results will be forthcoming, but it may take longer), or even the ability of the application to succeed. Under certain load conditions, it may be determined that it would be best to defer the execution since it is highly likely that it won't be able to complete.

Data Accesses: Data access refers to the data sources accessed (e.g., a particular database), the arguments used in the access (e.g., a query), and the items returned (e.g., the specific tuples). Types of data accesses of frequent interest are files, databases, and Web pages. Application behavior can be strongly influenced by file content. For example, the schema for an XML page is generally defined in another page containing a DTD or XML Schema definition. This may in turn be used by parsers to generate programming language (e.g., Java) objects, so in effect the data accesses may cause the generation of new components.

Expectations of Behavior: It is important to know if an application is behaving properly. This takes many forms, including constraints on (full or partial) results, quality of service measurements, resource consumption, or just a “feel” that the system is behaving properly based on experience with previous uses. Such information can be captured from specifications, the historical record of prior executions, and user feedback. For example, the amount of time required to produce a complex result might not be formally specifiable, but in practice may fall into a relatively small range. Similarly, a user may know that a certain data gathering activity usually produces a certain number of “hits” without being able to state precisely why this is so.

5. Software Surveyor

OBJS' *Software Surveyor* is a profiling toolkit to dynamically deduce and render the runtime configuration and behavior of evolving, component-based software. Information is synthesized from multiple sources and combined and rendered in a variety of formats and made easily accessible via the Web.

Software Surveyor addresses three distinct issues:

- What is the application doing?
- What is it supposed to be doing?
- Is it doing what it is supposed to?

Software Surveyor requires limited prior knowledge of application connectivity and has the ability to dynamically deploy probes, allowing its use with dynamically reorganizing applications and those lacking complete specifications. The next two subsections discuss the design of *Software Surveyor* and its current implementation status.

5.1 Architecture

The toolkit uses a three-tiered architecture (Figure 1) for runtime application analysis. The tiers correspond to data generation, data dissemination, and data consumption/analysis. The *probe management infrastructure* (data generation layer) manages and deploys probes to collect a variety of information from the running application and its environment. The probes pass on the collected information in the form of events to the *distributed event infrastructure* (data dissemination layer). This layer is responsible for relaying the events to interested subscribers, called gauges, which are part of the *gauge infrastructure* (data consumption layer). The gauges merge the event/information streams and make sense of it. In addi-

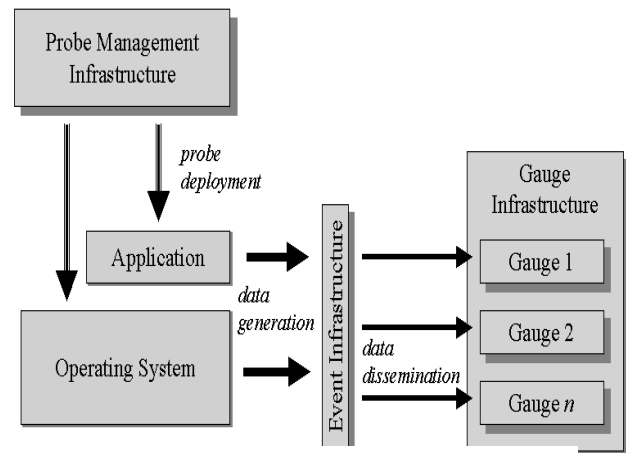


Figure 1. Software Surveyor

tion, results of this analysis are aggregated to identify “behavioral norms” to augment incomplete performance specifications. Finally, the probe infrastructure and behavioral norms can be used to signal users when the system is operating anomalously.

Probe deployment involves the insertion of probes, statically or dynamically, into data streams and execution flows so they can monitor and report on activity. The probe insertion technique will vary depending upon what is being instrumented, and when it can be or must be instrumented. For example, some probe types (source code probes) require precompilation-time insertion, while other probe types can be inserted into binary or bytecode at pre-runtime, or possibly during runtime.

Probe management is required for activation, deactivation, static and dynamic configuration, and probe removal.

Once deployed the probes generate typed information streams. We have adopted a basic event classification scheme that includes descriptors for type, subtype, and probed component name. This information is used by the event infrastructure and gauges for filtering and processing. Data emission rates may depend upon the type of probe, how it is configured, and the amount of activity at the insertion point. For example, one environmental probe (e.g. monitoring system activity) may be configured to emit resource utilization every 5 seconds, while an application probe (e.g. installed into the execution flow) may emit data whenever the method it instruments is invoked.

The data emitted by each probe is sent as events to the distributed event infrastructure so that it may be disseminated to interested gauges for analysis and visualization. The event infrastructure will accept data from any probe source, allowing *Software Surveyor* to support compatible third party probes.

Gauges subscribe to events by specifying event types or specific qualifying values or conditions based upon attribute values contained within the events. They may subscribe to multiple event types. Once a gauge receives events it may combine data from multiple event streams, perform analyses, synthesize new events, render visualizations, send feedback to the probes, and pass on information to other gauges.

5.2 Current Implementation Status

The current version of *Software Surveyor* includes a rudimentary probe management infrastructure including a probe insertion tool and several types of probes; a distributed event server; and several gauges.

In the initial version of *Software Surveyor*, we have created two types of probes:

- *AppliProbes* are used to instrument the application, and
- *EnviroProbes* are used to collect data from the operating environment.

Our current set of AppliProbes is implemented in Java. They are used to instrument both the target application and the underlying Java core class library. The probes generally emit method invocation arguments and related environmental data such as stack traces, invocation time, and thread information. Customized probes may emit other variables as well.

A subset of these are precompilation-time probes requiring insertion into source code. They are generally specific to the target application and are used to extract information that could otherwise not be acquired. We have also directly instrumented several core Java library classes to monitor application-environment communication, such as File, URL, and I/O stream access. These instrumented classes can be reused for any Java-based application monitoring.

Another subset of AppliProbes used are bytecode probes, which are inserted directly into Java bytecode at pre-runtime, and to a limited extent during runtime. Bytecode probes are inserted using a tool called the Java ByteCode Instrumentor (JBCI).

The *Java ByteCode Instrumentor* (Figure 2) automates the insertion of probes and probe stubs into Java bytecode. JBCI modifies .class files by inserting bytecode using customizable instrumentation techniques. JBCI can be extended with new probes and instrumentation techniques. Probes implemented in other languages can be called via JNI.

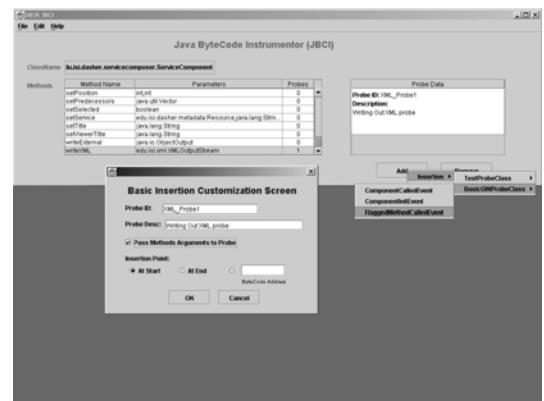


Figure 2. Java ByteCode Instrumentor

Our current EnviroProbes call upon operating system utilities to gather information on system status and resource usage. They monitor system-wide CPU utilization, application CPU utilization, and TCP bandwidth. They generate events at discrete configurable intervals.

While the current version of the probe infrastructure relies heavily on manually inserted probes, the next version will support on-the-fly probe insertion into running Java programs.

The probe-generated events are distributed by the SIENA Event Distribution Infrastructure [7]. SIENA uses a hierarchical distributed server architecture enabling the instrumentation and monitoring of distributed applications. *Software Surveyor* gauges subscribe to and receive events from SIENA.

The current set of gauges include Coalescer, EventMonitor, EventMerger, StackTracer, Historian, and Mapper. Once they receive events from the event server, which contain XML-formatted data, the XML is deserialized to first class Java objects for direct manipulation (Figure 3).

Coalescer merges streams of separately collected event information and renders this information on a timeline chart, performing limited aggregation of events by time

overall activities of each probed component in the application.

StackTracer converts streams of application events into a trace of program execution and emits an XML representation. The events emitted by a probe may be generated via several different execution paths involving the probed method. This gauge provides insight into frequency of invocation along each path. It can also be used to filter out paths (and therefore events) so that particular application behavior can be isolated for further analysis.

Historian archives execution traces and computes statistics of behavior.

Mapper provides a visualization of the time-based relationships between events of an application.

Software Surveyor v1.0 is implemented in Java 1.3 and has been tested under Windows 2000. v1.0 requires SIENA for event distribution. *EnviroProbes* is currently available only on Win2000 and WinNT.

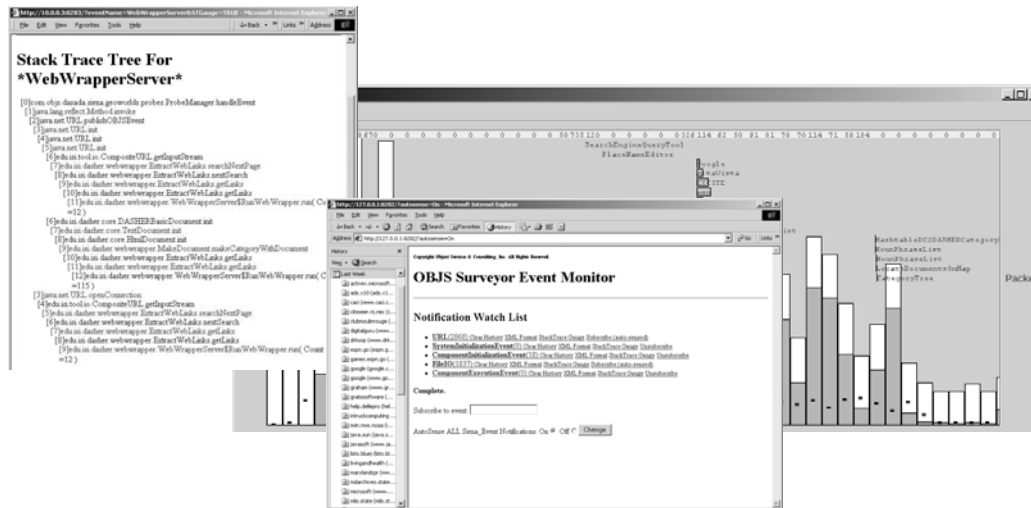


Figure 3. Software Surveyor Gauges

interval.

EventMonitor categorizes events by type and renders HTML- and XML-based displayable summaries with expandable detail. EventMonitor includes a web server to support browser-based access. It can be configured to subscribe to any subset of, or all, published events.

EventMerger, an extension of EventMonitor, performs event unification prior to rendering. Event streams may report on the same activities, but at differing levels from within the application. EventMerger identifies related streams of events by analyzing event content (e.g. stack traces, event type/subtype, component names and other attribute values). This can help, for example, to view the

6. An Example

We now illustrate the use of *Software Surveyor* in the construction and maintenance of a continuing application model. The selected application, GeoWorlds [8], is an Internet information tool that allows intelligence analysts to define “scripts” to locate, filter, and organize collections of Web-based information. GeoWorlds is representative of a large class of loosely coupled, highly distributed applications in which exact configuration and behavior cannot be specified a priori. Third-party components are heavily used, configuration and data access decisions are

made at runtime by tools outside the direct control of the application, and data sources may move, appear, become unavailable, and change their schemas without notice. As such, runtime monitoring and evaluation of behavior is necessary to provide analysts with high level, comprehensible support for determining whether a script is likely to work, whether a script is executing properly and making reasonable progress, and whether an information collection is plausible [9].

Portions of GeoWorlds can be modeled statically. Eventually, the model (both the static and dynamic parts) will be represented in the Acme ADL [4]. Representing all aspects of the model in this common, mature modeling language will allow the use of existing visualization tools and will enable the use of architecture analysis tools to ensure that the dynamic behavior meets static constraints. In the following, significant components and activities appear in *italics*.

GeoWorlds software consists of a *core* that resides in a JVM and an extensible collection of *external services* that may be in a variety of languages and resident anywhere, including within the JVM containing the core. The services manipulate a collection of *data sources* (mostly Web pages) using scripts to produce *InfoSpaces*.

The GeoWorlds core consists of a *ServiceComposer* for graphically writing scripts and several *job pools* for scheduling services accessible via different technologies; e.g., RMI, CORBA).

The external services conform to a static architectural pattern (a DAG of services); the leaves are *Web search engines* that locate Web content, the internal nodes are *data manipulation services* that filter, extract content and organize collections of Web pages, and the roots are *visualization services* that provide different views of the InfoSpace. Information flow between nodes is encoded as XML pages describing the InfoSpace and documents as processed up to that point. Each service has input and output schema to which they must adhere in order to function properly. Scripts are checked for sanity when they are constructed to ensure that input and output schema requirements are met, but because services can change their schema without notice, this is not always accurate.

Data items of interest are the *Web pages* manipulated by GeoWorlds (both source and pages constructed to represent the structure of an information space), and ancillary *data sources* (databases and files) accessed by services as part of their own operation (e.g., a database mapping place names to lat/long coordinates); these are not known to GW and are a key source of bottlenecks and errors if bound incorrectly or unavailable.

The static model of GeoWorlds as described above provides the “shape” of the application and identifies the kinds of components and activities that must be modeled and monitored dynamically. However, it is clearly

incomplete, since the services connected together into a script are written by analysts in the field and the pages accessed, how they flow through the script DAG as they are filtered and organized, and the behavior of an execution cannot be known until runtime.

Probes were embedded into the application to collect information about events identified by static analysis as being relevant; these included service start/stop, URL accesses, file accesses, and various initialization events. Whenever a probe detected such an event or condition, it was published as a SIENA event that could be subscribed to by (remote) gauges. The Web-enabled *Software Surveyor EventMonitor* and *EventMerger* gauges (Figure 4) subscribed to these events and created summaries and stack traces of application activity that lead to these types of events.

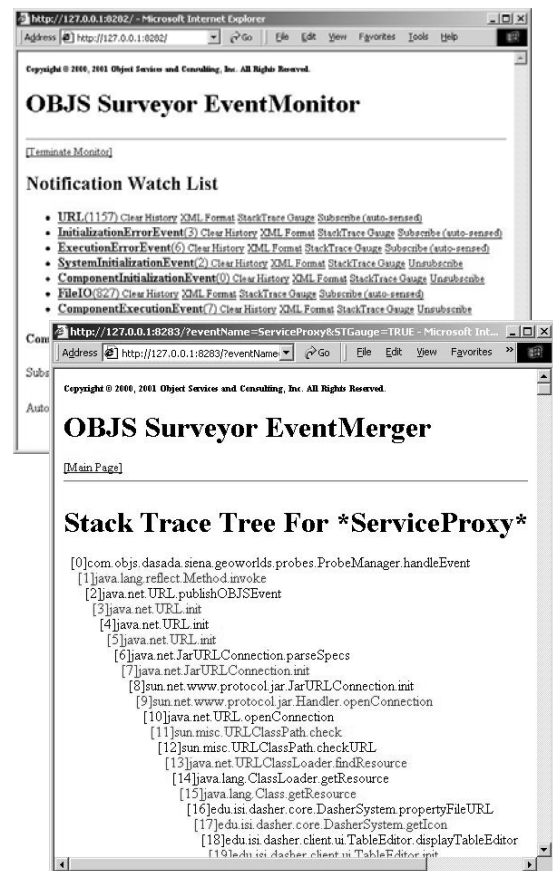


Figure 4. Event Monitor & Event Merger Gauges

Probes were also attached to the environment to periodically sample resource utilization of the GeoWorlds application and competing background activity. This information was subscribed by the *Software Surveyor Coalescer*, which also subscribed to a subset of the application events. These event streams were woven together to associate resource utilization, URL access, and

The screenshot shows a network traffic analysis tool interface. The top bar indicates the application is running on a Windows 7 system. The main window displays a packet capture of a DNS query. The packet is a standard query (type 1) from 192.168.1.100 to 192.168.1.1. The query contains a single question for the domain 'example.com' with type A. The packet is 100 bytes long. The tool interface includes a packet list on the left, a packet details pane on the right, and a packet bytes pane at the bottom.

Finally, information produced by the *Coalescer* over several successful and unsuccessful script executions was used to make a first-order approximations to the application’s normative behavior under particular environmental and scripting assumptions.

GW CPU Use Scatter Plot

The plot shows CPU Use (Y-axis, 0 to 3500) versus Time (5 sec intervals) (X-axis, 0 to 300). The legend indicates four data series:

- SUCCESS (dark grey)
- SUCCESS (light grey)
- FAILURE (black)
- 75% Load (light grey)

The SUCCESS (dark grey) series shows a sharp increase in CPU use, reaching a plateau around 3000 after 150 intervals. The SUCCESS (light grey) series shows a gradual increase, reaching about 2500 after 250 intervals. The FAILURE series remains near zero. The 75% Load series shows a steady increase, reaching about 1500 after 250 intervals.

Monitoring technology will be widely used only if its use is easy. For *Software Surveyor*, this equates to the processes of placing and activating probes, setting up the event distribution infrastructure, getting gauges to subscribe to the events emitted by the probes on the application, and providing convenient viewers for the gauge output.

Setting up the event infrastructure is easy. One or more SIENA event servers are started to disseminate events signaled by the probes. In addition, each *Software Surveyor* gauge uses a Web server to make gauge results available.

Finally, gauge output is readily viewable anywhere via HTTP using standard browsers.

Probe Infrastructure Improvements. Over the next year, we plan to improve the performance, coverage and flexibility of the probe infrastructure.

22

events; this will be rectified. Depending on the level of abstraction, an application may emit thousands of events per second. This places an excessive load on the event distribution layer, particularly if events are being subscribed over a WAN. We are working with the SIENA developers and the DASADA Event Infrastructure WG to develop a caching scheme that will allow events to be batched and transmitted in a group. This is more complicated than it sounds, since simple batching until a given number of events are collected may result in unacceptable delay in delivering events. Thus, transmission must be timed as well as batched. Further, because different subscribers may consume events at different rates, batching may be forced to use the lowest batching factor and therefore become useless. We will also add the ability to focus the attention of probes on areas of interest within the application. The notion of “focus” is discussed further below, but from a performance standpoint, the intent is to improve performance by reducing the number of events collected. This is essential, since no matter how fast the event distribution infrastructure is made, it will be possible to generate enough events to overwhelm it.

Coverage will be improved by better support for third-party probes and gauges and convergence toward a common event schema. Probes implemented in other languages can be called via JNI from Java probe stubs, and probes to monitor applications in other languages are supported by SIENA’s Java and C++ interfaces. Events are currently encoded in XML by many DASADA projects, but every project uses its own schema. A decision was made at the start of the project to defer the definition of a standard schema until more experience was obtained; a first cut at a common schema will be made in the next year.

Adding programmatic interfaces to the probe infrastructure will increase flexibility. This will allow activating/deactivating probes and dynamic probe placement to expand coverage of an application as it evolves. The next version of JBCI will support on-the-fly probe insertion into running programs (without any source code access) to support dynamic focus - evolving JBCI into a more fully capable Probe management tool.

Focus. *Focus* is the ability to concentrate attention on a particular aspect of the system being modeled. Focus has several aspects.

First is presentation; limiting the amount of information that is *presented* to a user so that the information presented can be used more easily, in essence trying to eliminate information overload. However, if the consumer of monitored information is a gauge, this becomes a non-issue.

A second aspect of controlling focus is the ability to limit the kinds and amount of information *collected* in order to avoid placing excess load on the event distribution

infrastructure and affecting application performance by signaling too many events. It is advantageous to place/activate probes only where needed to fill important gaps in the evolving model. For example, there is no use collecting information that cannot be used (either by gauges or to take corrective action) or that could equally well be determined statically (e.g., why dynamically determine that a connection uses TCP/IP if that was statically bound and not subject to dynamic change).

A third issue of focus is to address the issue of *incomplete probe coverage*. The collection of available probes determines how accurately an application can be modeled. Depending on the available types of probes and their placement, it may not be possible to profile all parts of a model adequately. Such areas are essentially “out of focus”, and given the lack of information, they must also be treated as immutable components. Security constraints may further prevent certain monitoring, even if the probes to do so exist. Thus, lack of focus may be involuntary.

In the next year, we will be developing a theory of focus and mechanisms to change focus. In particular, we need to be able to describe what is and is not known; lack of information about an event could mean that the event did not occur or that it was not in focus at the time it did occur and hence was not seen. We also need a way to represent focus in the event schema and in gauge outputs. Finally, we need to extend the probe management infrastructure to allow probes to report their focus (if possible) and to change the focus by inserting/removing probes, activating/deactivating them, and ordering them to collect different kinds of information. This in turn will require a better means of describing probe capabilities architecturally.

Model unification. Finally, we want to integrate dynamically collected information with static information and represent the combination in a single evolving model of an application’s deployment and behavior. We anticipate using xArch [3] for this purpose. In xArch, a distinction is made between the model of a family of systems and a specific instantiation of a member of that family. Not only would this provide a convenient modeling and display framework, but would enable a number of existing architectural analysis tools to be applied to dynamically gathered and modeled information. Both probe placement and event subscription assume knowledge of the general structure of the application and the kinds of events that a probe is capable of monitoring. Both appear to be amenable to simplification through the use of architectural models of the application (should such models exist). As noted above, two levels of architectural models can provide information about the structure and behavior of a family of systems and of a specific instantiation. It appears that there is sufficient information in these models to significantly ease the process of determining where to place probes. In the next year, we

plan to build a prototype tool to use ADL models to drive probe placement. Because this tool will then know where probes have been placed to carry out a particular monitoring activity, it can also inform the recipient gauges information about events to which they should subscribe.

8. Bibliography

- [¹] *Software Surveyor Project Homepage*, OBJS, 2001.
www.objs.com/SoftwareSurveyor
- [²] *DARPA DASADA Program Homepage*, Defense Advanced Research Projects Agency, 2001.
dtsn.darpa.mil/iso/programtemp.asp?mode=340
- [³] *xArch Project Homepage*, UC-Irvine, 2001.
www.isr.uci.edu/projects/xarch
- [⁴] D. Garlan, R.T. Monroe, D. Wile, "ACME: An Architecture Description Interchange Language," Proceedings of CASCON'97, Nov. 1997.
- [⁵] *Analyzing and Representing Componentware Structure*, David Wells, OBJS, 2001.
www.objs.com/DASADA/WhatsAComponent.doc
- [⁶] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable ObjectOriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.
- [⁷] A. Carzaniga. *Siena: A Wide-Area Event Notification Service*. University of Colorado Software Engineering Research Laboratory (SERL). www.cs.colorado.edu/serl/siena/
- [⁸] *GeoWorlds Project Homepage*, USC-ISI, 2000.
www.isi.edu/geoworlds
- [⁹] *IntelliGauge TIE Plan*, IntelliGauge TIE participants, Oct, 2000. [www.objs.com/DASADA/Intelligauge TIE.ppt](http://www.objs.com/DASADA/Intelligauge%20TIE.ppt)

(A-2) Measures of Success, OBJS Report, October 2000

An early version of success criteria for the project as a PowerPoint presentation.

Software Surveyor Measures of Success

David Wells

Object Services (OBJS)

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government

Kinds of Success

Software Surveyor probes, gauges, and infrastructure tools can be evaluated at several (increasingly meaningful) levels:

- Software Quality
- Probe & Gauge Coverage
- Gauge Precision
- Analysis Capability
- Task-Specific Evaluation
- Scenario-Based Evaluation

Software Quality

The quality of Software Surveyor probes, gauges, and ancillary tools can be evaluated through use by outside groups:

- Supporting software will be externally used by Columbia, WPI, BBN, and USC/ISI.
- Gauges will be demonstrated in the context of the GeoWorlds demo in May 2001.
- Gauges will be applied to typical bugs reported on the GeoWorlds Bug Reporting List.

Probe & Gauge Coverage

Probes and gauges can be evaluated by how well they perform their intended task.

- How completely and accurately can the gauges map an application's changing configuration?
 - a function of the ability to place probes at component boundaries (which is in turn dependent on the ability to probe in various technologies, collect the required information at these points, and deal with security restrictions that might preclude reporting).
 - in Y1, we will only capture information within the Java runtime; additional probing of DDLs and CORBA will be done in future years.
- Given that a complete configuration graph may be impossible to construct, how well can the gauges identify and address uncertainty in the graph?
- Is the level of completeness and accuracy that can be achieved for a configuration graph useful to an administrator or user?

Gauge Precision

The amount of detail that a gauge can provide is an important measure of the potential usefulness of the gauge, since w/o knowing how and why a configuration choice was made, it is difficult to determine if the choice is desirable or how to fix it.

- Between components within processes (fine grain - narrow scope) & between processes (coarse grain - wider scope).
- The process by which the connection was made
 - identity of the entity(s) that created the connection (linker, HTTP, CORBA ORB, Trader, manual, ...)
 - arguments used in creating the connection
 - source for the arguments (function call, file, ...)
 - how were "open point" arguments resolved? (i.e., to what values)
 - is the connection static or dynamic?
 - when was the connection made & modified?
- Whether & how the connection has been used.

Analysis Capability

Software Surveyor will provide analysis tools to compare configuration graphs and to match reified configurations to design specifications.

- Is it possible to match graphs so that corresponding components fill the same roles in both graphs? I.e., can matching be done preserving component roles as well as graph topology?
- Is the matching accurate?
- Can matching be performed when portions of graphs are unknown?
- How fast is the matching as a function of graph size? Is it fast enough to be useful?

Task-Specific Evaluation

Software Surveyor gauges can be evaluated based on how the information they provide facilitates certain specific software maintenance and debugging tasks:

- Improved diagnostic & debugging for multi-technology distributed software.
Goal = 75% reduction in time to identify configurations and activity patterns.
- Increased ability to evolve distributed software.
Goal = provide 75% of detailed configuration & usage status info needed by evolution planners.
- Low development & runtime overhead.
Goal = automatic or GUI-enabled insertion & 1% runtime penalty
- Reduced component footprint
Goal = 10-90% reduction in size of component footprints by identifying unused libraries or portions thereof (applicable only when such excess footprint exists)

Scenario-Based Evaluation

Software Surveyor success will be measured by how well it, in combination with other DASADA gauges, can improve the lifecycle behavior of a complex, distributed application. The GeoWorlds intelligence-analysis application is already in use at PACOM and improvements to its lifecycle behavior can be measured against historical data. Specifically:

- How *efficiently* GeoWorlds can be installed in different environments and its services deployed.
- How *easily* complex information management tasks can be scripted with assured semantic and syntactic interoperability.
- How *reliably* the scripts can be executed while maintaining desired quality.
- How *dynamically* the scripts can be evolved based on resource availability and requirement changes.
- How *efficiently* new services can be added to GeoWorlds while maintaining compatibility

B. Technical Reports

(B-1) Survey of Existing Instrumentation Tools, Paul Pazandak, OBJS Report, October 2000

A review of Java tools for monitoring applications and their environment as a starting point for new probe technology development.

Instrumentation / Profiling Software

Paul Pazandak
Object Services & Consulting, Inc.

Given the need within the DASADA project for software-based instrumentation tools that we can use to insert (**Java-based**) probes, I performed a reasonable scan on the internet for available systems. Using this list I derived a classification system (hence it could evolve) to help describe the available systems. We make this available for informational purposes and for anyone interested in (java-based) instrumentation and profiling software. If anyone wants to contribute, please forward references/reviews of the given systems (tools written in other languages welcome). [Disclaimer: As probe tools are secondary to our contract the reviews are not intended to be comprehensive in nature.]

Classification Scheme

I have divided the tools into the following categories and sub-categories:

- **Environmental** - (or application-external) these tools instrument an application without any application-specific code alteration through indirect means.
 - **OS / JVM level** - a special case whereby the underlying OS is instrumented
 - **External Application (Non-OS) level** - these tools provide application-external means to probe the application, such as instrumenting the communication path by use of a web proxy
- **Application-Internal** - Tools which enable one to instrument the application directly.
 - **Source Code level** - Tool supports instrumenting of the source code
 - **Bytecode level** - Tool supports instrumenting of the byte code / binaries. In general, these tools do not supply probes, rather they provide the *ability* to instrument given that they enable one to edit bytecode.
 - **Canned/closed systems** - The differentiation here is that these vendors provide a canned tool to instrument. That is, a constrained ability to instrument applications with their probing code, not yours. These tools come with visualization tools tool.

other categories??

[Some systems fit into multiple categories.]

Java-based Systems/Tools

Environmental

External Application

- [NetCool](#) - **Micromuse** - COTS - Large suite of software monitors and testing tools.
- [RMON](#) - **AdventNet** - COTS - Network Monitoring

JVM

- [Jinsight](#) - **IBM AlphaWorks** - ROTS - instrumented JVM
- [eTective](#) - **AverStar** - COTS - instrumented JVM. Apparent ability to target specific components & points of interest. Support for Visibroker CORBA, COM based applications, and some web servers.

Application-Internal

Source Code

- [JavaScope](#)^{NA} - **Sun Microsystems** - *Discontinued/Unavailable* -- Provided tool to instrument application and browser to view resulting data. Appears that it would instrument everything, no control over instrumentation techniques, location, or ability to add probes.

Bytecode (ROTS unless otherwise noted) Each provides at least a basic ability to modify java bytecode.

- [The JavaClass framework](#) (version 3.3.3) - **FU Berlin**
- [JOIE: The Java Object Instrumentation Environment](#)^{NA} - **Duke.edu**
- [BIT: Bytecode Instrumenting Tool](#)^{NA} - **Colorado.edu**
- [Binary Component Adaptation for Java](#)^{NA} - **UCSB.edu** -
- [JITrek](#) - (COTS) **Compaq**
- [CFParse](#) - **IBM AlphaWorks**
- [Jikes Bytecode Toolkit](#) - **IBM AlphaWorks** - I found this to be the most full-featured (albeit largest as well) bytecode editor

Canned Commercial Tools

- [JProbe Java Performance Tools](#) **KL Group Inc.** - A suite of performance analysis tools.
 - [NuMega DevPartner® Java™ Edition](#) **Compuware** - Canned application for application profiling. Doesn't appear to support user-defined probes or probe placement
 - [OptimizeIt!](#)
-

C++

Environmental : Application-Internal

Source Code : Bytecode

- [EEL: An Executable Editing Library](#) - **Wisc.edu** - Solaris-based
- [etch](#) - **Washington.edu** x86 source code
- [Instrumented Connectors](#) - **Teknowledge** - WinNT. Supports modification to the in-memory version of the program (no modification of the binaries on disk).

^{NA} - Does not appear to be an active project / tool

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government

**(B-2) Probe Architecture & Functionality, Paul Pazandak, OBJS,
December 2000.**

A categorization of probe capabilities, architectural forms, and implementation options.

Probe Architecture & Functionality

Paul Pazandak, Object Services & Consulting, Inc.

While probes are not the focus of the DASADA contract, they are the unseen enablers. And while we need not glorify them, we need to respect them, to worship them, and to understand the role they play -- without them gauges are just news reporters without news. While there are no really interesting research issues to solve here -- the probes require yet another intermediary infrastructure -- the important goal becomes interoperability. To begin with, then, we need a shared understanding of what a probe can be, its functionality, and its operating environment. The purpose of this exercise is simply to elaborate on the variables so that those interested can think about the functional and architectural design choices, as well as to seed a discussion so we don't have to deal with massive code changes (or at least reduce the changes necessary) at integration time.

While not mandated, it is assumed because of the significant benefits (and some mention of it), that a probe management infrastructure will be used. While this may dictate at least a few of the design decisions (e.g. such as the notion/use of *probe stubs*), several decisions remain -- such as the division of responsibility between the management infrastructure and the probes. Other decisions are more probe-centric, relating to control of output generation, probe placement, and invocation.

[Note that at a given level in the tree if alternative choices are listed they appear with bullets, if the level lists subcategories then no bullets are used.]

- **Probe Type** - Two probe related types: probe stubs and probes.

Self-contained Probe - The probe is inserted directly, and multiple probes at one insertion point require each to be inserted independently of each other (one after another).

Insertion - When is the probe installed

- **Static** (e.g. compile time / load time)
- **Dynamic** - If tools permit. It is possible to support dynamic probe insertion and removal, though the degree to which this is possible is constrained & dependent upon the programming language. In Java, if the class being modified has no current instances or has not been loaded yet, then runtime modification is straightforward. If class instances exist then more complex memory-based tweaking techniques will be required (e.g. as used by interactive debuggers like Visual Cafe). Dynamic insertion and control are simplified greatly by the use of probe stubs.

Execution - Execution order is determined by relative placement of probes in the flow of control

Probe Stub - One or more self-contained probes *plug in* to a probe stub. This facilitates runtime probe insertion and removal. Only the probe stubs need to be inserted into the application code. At runtime, one or more probes can be associated with a given stub using a registration API. When the stub is invoked, it in turn invokes each of the registered probes. This approach also enables a potential flow of data between a set of registered probes. Issues regarding data flow between probes and data flow between gauges are addressed in the **Other Issues** section. How the stub determines what data (e.g. parameters, local/global state) to pass to each probe, and how it passes the data must also be addressed.

Management - some process needs to manage the list of 0+ active (and inactive) probes associated with a stub

- **Local Management** - Stub is aware of and manages the list of probes that are registered with it (the stub has a management API which supports probe registration). When invoked, it locally manages the execution of the registered probes.
- **Management Facility** - The stub calls out to another facility when invoked and that facility in turn controls the execution of probes registered with that stub

Execution (if 2+ probes) - When more than one probe is registered with any given stub, execution order and data passing must be addressed (different execution orders may produce different results).

- **Independent / Parallel** - the registered stubs are executed in parallel/independently (this would appear to be acceptable as long as the probes do not modify application/environmental data)
- **Dependent / Series** - the registered stubs are executed in some defined order (esp. necessary if the probes modify application/environmental data)

Stub Insertion - When is the probe stub installed

- **Static** (e.g. compile time / load time)
- **Dynamic** - if tools permit

Probe Insertion - The stub facilitates runtime insertion **and** removal.

- **Probe Existence**

Lookup - How do probe data consumers locate the probes they are interested in?

- **Static Binding** - Gauges and other consumers simply assume that the probes they are interested in exist and are running
- **Dynamic Binding** - Consumers need to lookup and potentially cause the activation of the probes. Runtime binding is required.

Advertisements - It is possible that probes advertise themselves, e.g. in a lookup service. This assumes that one can describe in an accurate way the probe, what it is monitoring, and its capabilities. It also requires a facility for ads.

- **Offline** - Advertise for use & insertion
- **Runtime** - Advertise that they are running and available
- **Probe Security Issues** - Can anything register to get data from a probe? What if the data is classified or needs to be protected?
- **Probe Behavior** - Probes may be designed for several different types of tasks.

Data Examiners - The range of data that the probe monitors. The events it generates may change based upon the data content.

- **None** - The probe simply executes & does not inspect any data (e.g. used to report that a method has been called)
- **Parameters** - The probe inspects parameters passed into the local state (method)
- **State** - The probe inspects the global state of the application
- **Environment / External** - The probe inspects environmental or other application external data
- **Runtime configurable** - What the probe inspects may be altered at runtime (controlled either directly via the probe or via a management facility)

Data Passers - The probe may pass data available to it out to its consumers. The format of the data passed is a separate, but related, issue.

- **Parameter** - The probe passes out some subset of the parameters passed in to the local method in which it is installed
- **State** - The probe passes out some subset of the (global) state available to it
- **Environment / External** - The probe passes out some subset of the environment

- **Runtime configurable** - What is passed out by the probe is runtime configurable

Data Manipulators - The probe or possibly its consumers which it has sent the data to alter some subset of the data (the types listed above). The probe needs to ensure that changes are propagated correctly

- **Runtime configurable** - What is modified is runtime configurable
- **Probe Management** - Overall control of the probe, e.g. controlling whether it is active or inactive.

Self-managing - Regulating the probe requires direct interaction with it for all management features (via a probe management API)

Centrally-managed - Regulating a probe requires interaction with a management facility.

- **Pull** (polling) - The probe polls the facility for control information
- **Push** - the facility updates the probe via the probe's management API

Divided - Management is divided between the probe and a management facility (non-overlapping)

- **Probe Execution / Placement**

Execution -

- **Control Flow** - Probe is inserted into program code & naturally executed based upon the flow of control of the application. The probe will have access to local state, method and class state, and any global state.
- **On Demand** - Probe is not in any flow of control, and is executed only on demand (when invoked). Exact placement will determine the application state that these probes have access to (e.g. class and global state).
- **Event Driven** - Probe, like a gauge, registers for events and is invoked when those events arise. Exact placement will determine the application state that these probes have access to (e.g. class and global state).

Placement (Specification) - How is probe (stub) placement specified?

- **Descriptive Language** - Some means to accurately describe this information should be used (e.g. such as a probe placement specification language, or extended ADL).

- **Manually** - An alternative is to simply manually place them, but this is a more limiting solution since they will need to be manually placed in each successive version of the instrumented application.
- **Probe State** - A probe may have a persistent state it wants to maintain

Stored/Managed Locally (by Probe) - The probe manages this state

Managed Externally - Another facility maintains the state for the probe

- **Probe Output**

Data - The probe generates data / events reporting information it has been programmed to generate. Options include who the information is sent to, *who* sends it, *how* it is sent, *when* it is sent is, and in what format.

Listeners / Registrants - The list of recipients for broadcasts from a probe

- **Management Facility** - A management facility manages the list of registered listeners
- **Probe-managed** - The probes manages the list

Delivery - Who delivers the data to the consumers

- **Management Facility** - The probe sends the output to a management facility which then sends it to all registered consumers
- **Probe-managed** - The probe manages which consumers receive data (and possibly their scopes of interest)

Transmission - If probes send events synchronously to their consumers it can affect performance, and even stall an application if the consumers are not available. Of course, this may not always be possible -- e.g..if it is a probe stub sending data to a probe which modifies the data, then it must be sent synchronously so the data can be updated before application execution continues.

- **synchronous**
- **asynchronous**

Timing - Controlling when the probes output the data they have generated

- **Autonomous (every time they are invoked)**
- **Controlled (e.g. off/on)** - We can control data output by activating or inactivating the probe
 - **On/Off State Stored locally** - The probe locally stores its activation state

- **Calls out to management facility** - Before any execution the probe checks another facility for its activation state
- **On state-change only** - The probe only reports differences (requires that it has access to historical data, whether locally or remotely maintained)
- **Constraint-based** - Output is determined based upon constraints, either internally or externally dictated.
 - **Internal**
 - **External** - e.g. a management facility
 - **Runtime Configurable** - the constraints can be altered at runtime by consumers, the probe itself or some other object
- **Pollable** (pull) - The output is retrieved via polling of the probe (suitable when the probe is not otherwise executed by natural flow of control)

Format - The format of the data output by the probe

- **Static** - Preset
- **Adjustable (e.g. based upon specific event or consumer)**
- **Externally formatted** - output to another consumer which then formats the data as required for its intended recipients (essentially this is the same as static)

APIs - Given the functionality mentioned above it is certain that both the probes/probe stubs and the management infrastructure will have APIs for configuration/control. Both programmatic and GUI-based interfaces could exist, enabling code-based and direct user-based control of these objects.

Other issues

- **Probe Event/Data Correlation** - How does one relate events generated by different probes as belonging to the same unit of work / user / etc? This is more complex when the system is multi-user and/or multi-threaded.
- **Homeless Probes** - Not all probes may have an instrumented home -- a foreign application into which they are embedded. For example, environmental probes which provide information which is needed before an application is started (e.g. *"Is the car on a drivable surface? Does the car have safe tires installed? Is there a bomb attached to the car?"*). In such situations the probes may exist in specialized applications built just for them.
- **Namespace / Probe placement specification language** - To register presupposes that gauges know how to describe the probes whose events they are interested in, and how to request that specific probes be installed & activated.
- **Probes vs. Gauges** - What is the difference between a probe (if stubs are used) and a gauge? Gauges register for and consume XML-ized events from probes. Do probes and probe stubs communicate with other probes using XML-ized events, or higher level Java object representations? Some different perspectives:

- o **Efficiency** - Of course, if probes have to parse XML it will be slower than passing java objects, especially if all probes/stubs are in the same local process.
- o **Pass by reference** - If a probe needs to modify data, then pass by value (e.g. XML events) complicates things a bit.

If probes never modify data, we could potentially view probe stubs as probes, and then view probes as gauges.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government

**(B-3) ProbeMeister: Distributed Runtime Software Instrumentation,
Paul Pazandak and David Wells, 1st Int'l Workshop on Unanticipated
Software Evolution (USE), Spain, June 2002**

Abstract: Dynamically deployable software probes facilitate ad hoc runtime application monitoring and troubleshooting. Using the latest features of Sun Microsystems' JDK 1.4, we have built a prototype system called ProbeMeister that can attach to multiple remotely running applications, and effortlessly insert software probes to gather information about their execution. This information can be used to effect changes within the running applications to recover from unanticipated failures, or to improve their operation. While ProbeMeister is useful during software development and testing, its advantages are better realized after the software is up and running at the users' sites.

ProbeMeister

Distributed Runtime Software Instrumentation

Paul Pazandak and David Wells*

[{pazandak,wells@objs.com}](mailto:pazandak,wells@objs.com)

Object Services & Consulting, Inc. Baltimore, MD

Abstract

Dynamically deployable software probes facilitate ad hoc runtime application monitoring and troubleshooting. Using the latest features of Sun Microsystems' JDK 1.4, we have built a prototype system called ProbeMeister that can attach to multiple remotely running applications, and effortlessly insert software probes to gather information about their execution. This information can be used to effect changes within the running applications to recover from unanticipated failures, or to improve their operation. While ProbeMeister is useful during software development and testing, its advantages are better realized after the software is up and running at the users' sites.

1 Introduction

Software probes enable the monitoring of running applications. Current probe tools are primarily designed for software testing: developers insert probes into their code or underlying OS during testing to emit data to help locate bottlenecks, memory leaks, bugs, or to visualize code coverage, etc. Probes of this kind may also be left in an end-user version of an application for bug reporting: if users encounter problems at a later date, the log files generated by the probes can be sent back to the software vendor for analysis. Both types of uses require skilled programmers to place and compile probes into the application and

to determine the corrective actions to be taken once the probe data has been gathered and analyzed. Required changes are made and the software is recompiled again.

Our goal in developing our own instrumentation tool was to produce a technology suitable for distributed reconfigurable component-based software - potentially widely-distributed applications whose components are loaded on demand. Such systems are difficult to extensively test prior to deployment, partially because their environment is often immense (think Internet scale) and constantly changing. Moreover, the components may be developed by separate companies and typically evolve independently of each other, increasing the probability that problems will arise.

To probe these kinds of systems, our tool would need to be able to connect to running applications and deploy probes to each of the distributed components, then gather up all probe output for runtime tool-based analyses. In conjunction with other tools, required changes would be made without recompiling or restarting the application¹.

Using the latest features of Java JDK 1.4, we have built ProbeMeister, our second-generation instrumentation tool capable of deploying probes into remotely running Java software. ProbeMeister instruments Java bytecode, and works without needing to copy supporting code libraries to the remote machines. Probes can be deployed and removed at will into any running Java application, remote or local.

* This research is sponsored by Defense Advanced Research Projects Agency and administered by the US Air Force Research Laboratory under contract F30602-00-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government

¹ Imagine needing to modify, recompile, and restart a vital army tank subsystem during battle.

ProbeMeister is being developed as part of the Software Surveyor project [^x] within the larger DARPA DASADA program [^{xi}], the goal of which is to develop technology to model, monitor, and manage dynamically composed and evolving systems².

2 Overview of ProbeMeister

ProbeMeister facilitates the instrumentation of a distributed running application with software probes². It accomplishes this in part by manipulating the in-memory representation of the running application. A key capability of ProbeMeister is that its extensible set of software probes can be inserted or removed at any point while the application is running. A second key capability is that it supports the instrumentation of multiple remotely running applications. Both are necessary for monitoring evolving, distributed applications; since the application's connectivity and components may change during execution, it is essential to be able to insert and manage probes in multiple remote components simultaneously.

Prior to developing ProbeMeister, we developed the Java Bytecode Instrumentor (JBCI). JBCI is a static bytecode instrumentor. It requires a multi-step process of loading a class off-line into JBCI, deploying and customizing the selected probe(s), saving the modified class, and then restarting the application. Removing a probe requires similar steps. What we found was that once we knew exactly where we wanted to deploy all of the

probes, the process was relatively quick. However, we also found that probe placement is an intensely iterative process unless perhaps the user is also the developer. For the overall project that we are involved in it is understood that a ProbeMeister user is not always the developer, but perhaps only a skilled application user having solid but general knowledge about how the application works [^{xii}]. When the application is not performing as expected, either as determined by the user or by pre-deployed (possibly even statically deployed) monitoring probes, task-specific probes could be deployed to gather more information to determine if a given component is not working as expected. A corrective response could be to modify specific parameters in the code to tweak its behavior, or to replace the component with a more reliable or more available one.

As software developers, we felt that the probe deployment cycle was a hindrance to ad hoc exploratory probing. This was the prime motivation for replacing JBCI with the much more dynamic ProbeMeister. In moving from JBCI to ProbeMeister, we immediately enjoyed the benefits of dynamic distributed probe deployment. Not only did it practically eliminate the deployment cycle, but also the results generated by the probes could be seen immediately without having to restart the application.

Supporting easy probe placement by non-developers requires additional tools and interfaces having knowledge of the application's architectural model, that can suggest probe deployment locations (or automatically deploy probes) based upon the problems the user (or the analysis tool) wants to troubleshoot.

² ProbeMeister also supports the dynamic creation of new classes and complete redefinition of existing classes.

Model-based probe deployment is the focus of another aspect of the Software Surveyor project, and will be the topic of a future paper.

Finally, ProbeMeister has not been designed to compete with coverage-oriented optimization tools. These tools are certainly more efficient at this since they can statically instrument an entire application with perhaps hundreds or thousands of probes to collect performance data. ProbeMeister is geared toward targeted placement of probes to inspect (and possibly effect changes upon) a running distributed application. However, the new interfaces in JDK 1.4 now makes it unnecessary to deploy any probes into a remote application to provide coverage feedback. We just haven't focused on exposing this capability in ProbeMeister yet.

2.1 Related Work

Prior to developing JBCI (about two years ago) we performed a reasonable search and tool review of several Java (and C++) bytecode related packages (see Related Works in section 8 for the links to the mentioned Java tools). We were looking for an extensible tool that would allow us to write our own bytecode probes and deploy them. While we could have looked at source code probe deployment tools, we didn't want to limit probe deployment to applications in which we had source code access, nor did we want the overhead associated with adding a recompiling step to the deployment cycle. We found that some of the available instrumentor tools appeared to be closed, and didn't allow one to write their own probes. These tools were geared toward software profiling (e.g.

JProbe, NuMega, OptimizeIt!). Some other tools were close to what we were looking for but were not extensible, no longer supported, or too costly (e.g. JOIE, Jtrek, JFParse). Yet another group approached instrumentation by providing modified or pre-instrumented JVMs (e.g. Jinsight, eTective, BCA). Finally, the last tools (e.g. BIT, Jikes, and BCEL) were simply bytecode editors. For our needs, we thought it would be more efficient to prototype our own tool using one of these editors. Jikes is the editor we integrated into JBCI.

Even though JBCI worked reasonably well as a standalone tool, it could not be used (in the next stage of the DASADA program) by other tools at runtime to deploy probes since it only supported static instrumentation. This was an obvious and significant drawback to using JBCI. Thanks to several people at Sun Microsystems, we were fortunate to get access to an early version of JDK 1.4 in which the Java debug interface (JDI) had been extended to support remote runtime bytecode modification [^{xiii}]. Using JDK 1.4, we began a complete re-implementation of our tool (about one year ago), now called ProbeMeister.

We should mention that we did find some similar tools once we began implementing ProbeMeister. For example, we found an interesting product called RootCause, which offered a sort of "one-time" dynamic probe deployment by instrumenting a class as it is loaded into the JVM. Other similar products like this exist in the C++ world, such as NTWrappers, that can modify a DLL just prior to it being loaded by the OS. Of course, none of these offered the kind of flexibility we desired.

3 ProbeMeister Architecture

In this section we present a high level view of the ProbeMeister architecture. In the lowest layer, ProbeMeister has a Virtual Machine (VM) Manager that accepts or initiates connections with other JVMs via the JDK's JDI interface. Connection behavior is enabled and configured by the targeted application through command-line arguments to the Java interpreter -- the application may initiate the remote connection (as a client) to ProbeMeister (running in a separate JVM), or it may begin its execution and allow ProbeMeister to initiate the connection (acting as a server) at some later time. In either case the targeted application requires no additional code as the underlying JDI extensions manage the connection to ProbeMeister. Using the JDI interface, an application like ProbeMeister can set breakpoints, subscribe to events (e.g., class loading, method entry and exit, etc), modify methods, and even create new classes ones on the fly.

If the application initiates the connection to ProbeMeister, the connection is established before the core JDK classes have been loaded, and therefore also before its main() method is invoked. This enables probes to be deployed before any of the application code has been invoked. It therefore allows the probes to capture the application's entire behavior from the beginning. When the connection to the application is opened, ProbeMeister stops the remote JDK's execution prior to loading of any of the application's classes. This enables the user to instrument any of the core JDK classes (such as java.io.File to monitor file access) and therefore capture all application activity. The user could also

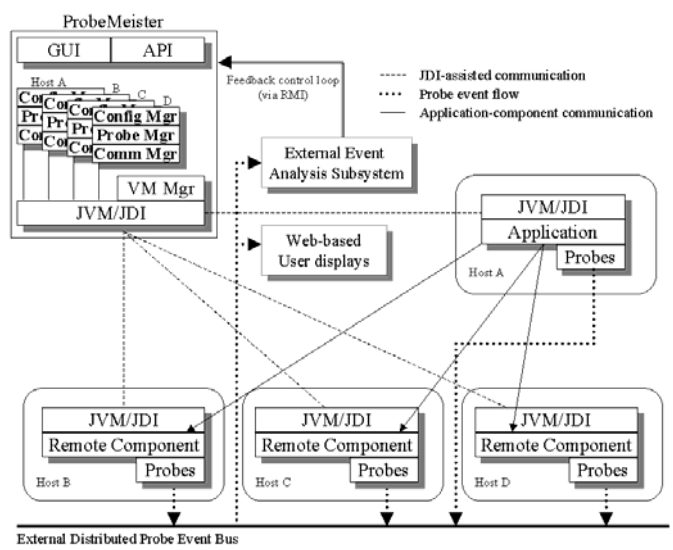


Figure 1. ProbeMeister Deployment Scenario

schedule probes to be automatically deployed as the application's classes are loaded.

While having each application connect to ProbeMeister is convenient, we feel that it is an unreasonable constraint. ProbeMeister should be able to be activated on-demand, as needed. JDI-specific Java interpreter command line arguments allows an application to accept a remote connection at any point during its execution. Using this approach, a ProbeMeister user (or external tool) may request a connection to an application by specifying its address and port (as defined in that application's command line arguments). After a connection is established ProbeMeister may be used to deploy probes.

Once an application is connected to ProbeMeister it is assigned a set of components: a Connection Manager that manages the communication with the remote application; a Probe Manager that controls the creation and deployment of probes; and, a Configuration Manager that provides control to deploy or remove several probes (a probe configuration) simultaneously.

Finally, in the highest layers, ProbeMeister provides a user interface as well as programmatic interfaces so other tools can control it (locally or remotely). The following describes each of the connection-specific components in more detail. Figure 1 shows the architecture of ProbeMeister in a typical deployment scenario.

In the depicted scenario, ProbeMeister is connected to four remotely running Java Virtual Machines (JVMs) that make up a given distributed application. Probes that have been inserted, when invoked, emit descriptive events to the Siena Distributed Event Server [^{xiv}] event bus³ for remote delivery to interested consumers. The emitted events are consumed by a separate external system (tools under development) that analyzes the events, generates user consumable output (status or warnings), and may also feedback into ProbeMeister by dictating further probe reconfiguration. Of course, ProbeMeister can be used for manual user-driven monitoring and analysis. For this, we have built web browser-based HTML and XML user displays that collect and categorize probe events, and display event details⁴

3.1 Communication Manager

When a connection is established between the VM Manager and a remote application the connection is assigned to a communication manager. The Communication Manager manages the connection with a distributed application or component via the JDK 1.4 JDI interface. It provides the routines for

accessing the remote classes, modifying the classes, and monitoring the state of the JVM. All calls affecting the remote JVM pass through this component.

3.2 Probe Manager

The probe manager controls the insertion and removal of application probes. Probes may be inserted when a class is first loaded, before any of its methods are invoked, or at any point after that. Insertion involves loading the chosen class' Java bytecode from its class file, modifying the selected method by inserting the probe-specific bytecode, and writing out the modification to the remote JVM using the JDI API call:

```
VirtualMachine.redefineClasses()  
s().
```

ProbeMeister currently uses the Bytecode Engineering Library[^{xv}] to modify Java bytecode⁵. To understand the minimum cost to deploy a probe, it takes on the order of 20 milliseconds to create a basic probe, modify the bytecode, and invoke `redefineClasses()` on a small locally running application. It took an average of about 250 milliseconds to deploy the same probe on the same application running in Baltimore with ProbeMeister running in Minneapolis⁶. This has been more than adequate to date given that we have created on the order of no more than tens of probes per remote application.

`redefineClasses()` takes as an argument the entire modified bytecode

³ Siena is the event bus for many projects in the DASADA program.

⁴ Probe event content may include probe location, invocation time, stack traces, user-assigned values, and method arguments, for example.

⁵ We switched from using IBM's Jikes because of licensing constraints (only evaluation licenses were available), and because it was no longer being improved

⁶ We found that it takes several minutes to define a new class which is a concern to us, but we have not yet studied this issue in detail.

of the class. Once invoked, it replaces the class definition in the remote JVM. However, only new invocations will execute the new version of a modified method; currently running invocations will run to completion using the older code. And while the JDI specification allows for considerable changes to the bytecode (e.g. new methods, new attributes, completely redefined class, etc), it is up to the individual JVM implementations. At this point, Sun's JVM implementation only supports method modification. Once ProbeMeister modifies a class, since the modifications are transient a copy of the modified class bytecode is retained and used as the basis for any further probe insertions or deletions. A detailed description of the supported probe types is presented in a later section.

While ProbeMeister's Probe Manager has been designed to support the management of heterogeneous (multi-language) probes, thus far we have focused exclusively on supporting dynamic (or runtime) Java bytecode probes. Runtime probes are inserted while the application is running, while static probes are inserted when the application is offline. Runtime probes are transient by default, and are lost once the application terminates; static probes are persistent by definition. To make runtime probes persistent, the in-memory modifications need to be saved back to disk in Java classfile format. The modified classfiles can replace the original classfiles, or be stored separately (however, the configuration manager eliminates the need to do this, as described in the next section).

Finally, while ProbeMeister maintains a list of inserted probes for each JVM, the Probe Manager is also

capable of automatically identifying all probes that have been previously inserted (whether statically or dynamically) by parsing and relatively quickly examining a method's bytecodes. This mechanism is also used to validate external configuration files to ensure that they accurately reflect the current set of inserted probes in a given instrumented version of the application.

3.3 Configuration Manager

While the act of placing probes is quite straightforward, it would become tedious if one had to redefine and redeploy probes each time ProbeMeister connected to the application -- for *each* remote component. For this reason we implemented a probe configuration manager. The Configuration Manager is responsible for tracking and recording all probe deployments to each application. The current configuration can be viewed and saved (to XML-based configuration files) at any point. Once saved, a configuration can again be viewed, and also reloaded and reapplied. Reapplying a configuration causes all probes to be reconstructed and then deployed to the selected application.

A second use of configuration files is to define probe sets that target specific activities or parts of the application (e.g. file access, network traffic, etc.). Using these sets, one could load and monitor the output from one probe set, then deapply the set (which removes deployed probes) and reapply another set.

3.4 User Interface

The graphical user interface (see Figure 2) provides access to all of the features described above. Virtual machines (applications) waiting to attach to

ProbeMeister are announced at the bottom of the display. As stated, the user may also initiate a connection (using the menus) to a remotely running virtual machine. Once connected, the user *resumes* the virtual machine's execution. Each tab in the display represents a different remote virtual machine running a separate component or application. This figure shows two applications that ProbeMeister is connected to. The first is a remotely running GeoWorlds^[xvi] client application, the second is a service component used by the client. GeoWorlds is a central part of the software testbed within the DASADA project because it is a distributed component-based application that dynamically assembles itself on-demand.

The interface lists all of the application's classes that have been loaded (the core JDK classes have been filtered out using the controls at right). The *add()* method has been instrumented with a simple probe -- this probe outputs a user-provided string to the application's console. From the list of classes one can also see a class that ProbeMeister has dynamically deployed (called OBJS_Breakpoint) into the remotely running virtual machine. ProbeMeister automatically deploys this class into each attaching JVM to control

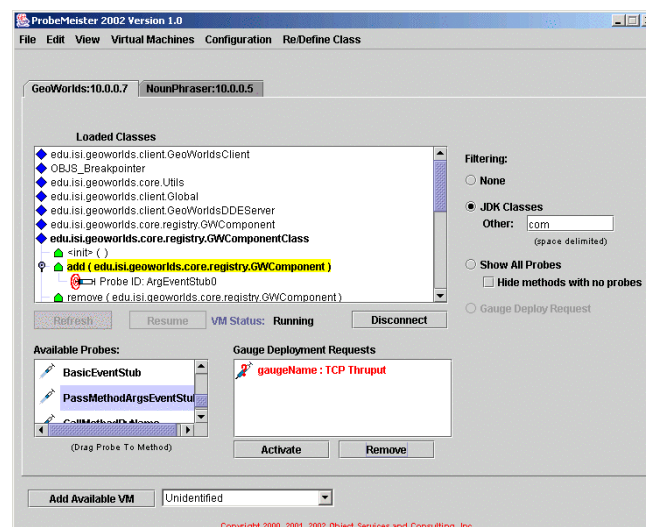
breakpointing (methods belonging to classes in a remote JVM can only be invoked at breakpoints).

Probes are inserted by dragging a probe from the list of probes onto the desired method. Most probes require some configuring and present displays for customization. The probe icons are used differentiate between deployed and undeployed probes, and simple probes and probe stubs (described later).

The Gauge Deployment Requests list illustrates how external tools may suggest deployment locations within ProbeMeister. These tools may also automatically deploy probes without user intervention. This interface is only present when requested (and is the subject of a future paper on Software Surveyor).

3.5 Other Interfaces

ProbeMeister provides access to its functionality through local and RMI-based programmatic interfaces. As seen in Figure 2 and discussed briefly above, Gauge Deployment Requests are sent over RMI to ProbeMeister. These external software gauges consume the events emitted (over the event bus) by deployed probes, so when gauges are first activated they suggest or auto-



deploy (via deployment requests) the probes required to monitor the targeted activity.

4 Probes

ProbeMeister provides a Statement Factory to generate bytecode probe definitions on the fly. Probes are defined like *recipes* where the ingredients are Java bytecodes. Defining a probe recipe requires identifying the series of calls to be made to the Statement Factory. Each call adds one or more Java bytecodes. While several probe recipes are provided, others can be added to the library by extending the BytecodeProbeInterface. Probes can also be constructed in an ad hoc manner by directly calling the Statement Factory via the programmatic interface. Furthermore, the Statement Factory can also be extended with more functional bytecode building blocks. The following example illustrates how the simple PrintStringProbe class creates bytecode using the Statement Factory and inserts it into a specified method (defined in a bytecode location - bLoc).

```
[a] StatementList sList=
    BytecodeMgr.createStatementList(bLoc);

[b] StatementFactory.createPrintLnStmt(
    sList, userStringToPrint);

[c] SimpleProbe simpleProbe = new
    SimpleProbe (probeID,
    probeDescription, probeType, sList,
    bLoc);

[d] BytecodeMgr.insertProbe(simpleProbe)
    ;
```

Initially [a], a new structure (StatementList) is created that will hold (and validate) the probe-specific bytecode. In [b], the call to the Statement Factory's createPrintLnStmt() generates bytecode that outputs the specified string, and then

inserts the custom bytecode into the StatementList. In [c], a new simple probe wrapper is created (it knows how to deploy simple probes). It is passed a unique probe ID, a probe description, a probeType (PrintStringProbe), the StatementList, and the bytecode location. Finally, the probe is inserted into the targeted method. Once this is done, redefineClasses() may be called to propagate the update to the remote JVM.

ProbeMeister defines two types of deployable probes: simple probes and probe stubs. Simple probes are self-contained units of code. While they may call out to other methods owned by the application, they do not require any more probe-specific code to function. The current set of predefined simple probe recipes include a probe that outputs a user-defined string (discussed above), one that outputs the method's argument values, another that calls a specified static method, and a similar probe that calls a static method using introspection wrapped with exception handling. Simple probes may output information to the console of the remote application (such as argument values), or modify method state, for example. But, without supporting code, a probe cannot emit events. This is one motivation for probe stubs.

As there is only so much one can do with a probe in a single method, we found a need for a probe that could be divided in two: we call them probe stubs and probe plugs. A probe stub, like a simple probe, may perform intra-method manipulations such as modifying argument values or outputting data to the console. However, a probe stub is also able to perform more complex tasks because it calls out to one of an array of

probe plugs. For example, two of our pre-defined stubs (probe recipes) include one that emits status information, the stack trace, a user-defined string, an event name and sub-event name; the other also emits the set of method arguments. This information can then be passed to a plug for further processing, and even return values back to the stub (e.g. to effect state changes).

Unlike probe stubs, which are written in Java bytecode, probe plugs can be written in Java. This really simplifies the writing of the bulk of the probe's functional code. A probe plug provides specific functionality that may perform any task. We currently use probe plugs to emit data from the probe stubs over the Siena event bus. Stubs are matched to plugs by their method signatures. When a user selects a probe stub to install, the Probe Manager returns a list of all compatible probe plugs from the ProbePlugCatalog. The user then selects an appropriate plug based upon its functional description. Like simple probes and probe stubs, new probe plugs can be added by registering them in the appropriate persistent catalog.

When stubs will be used, either the remote virtual machine must include the associated probe plug classes in its classpath, or ProbeMeister can port the probe plug classes to the remote virtual machine on the fly. The latter of course is preferable, as otherwise the plug code will need to be copied to each remote computer. However, if a considerable number of classes need to be deployed it may require significant overhead⁷.

⁷ The Siena Distributed Event Server is composed of 54 classes, making it more practical to copy the jar file to each site. However, it is likely that we could modify it to reduce

Finally, the Statement Factory validates the structure of each probe (only the Statement Factory can insert bytecode into a StatementList) and uses a wrapper mechanism to ensure that the probe can be removed once deployed.

5 Issues

There are a number of issues and limitations that are worth mentioning. First of all, as previously discussed, simple probe output is constrained to the remote JVM's console window because the probe code is executing within the context of the remote application. This is useful for certain types of debugging and monitoring, especially if the application is local. But, if the application is distributed, there must be a way to collect the probe output from each remote JVM. Using probe stubs and supporting code a probe can emit events external to the remote host. As previously mentioned, we currently support this capability using Siena. The events generated by the probes are published to a remote Siena event server and subscribed to by our user-oriented Siena event monitor (and other Software Surveyor gauge tools), which then displays the event data in a web browser. Other event publication schemes are also possible. For example, one could use the Java JDK 1.4 Logger API to emit probe events in the form of log messages via TCP streams to a remote collection system.

Another issue is probe control. Currently probes deployed in the remote application can only be disabled by removing them. One potential alternative would be to simply modify the probe bytecode by inserting a jump instruction

the number of classes significantly, thus making it possible to deploy on the fly.

to bypass the probe code. This is slightly more efficient than removing the entire probe and reinserting it at a later time. Another alternative would be to port a new class that contains a vector of Boolean switches. Each probe would then check its own on/off value in this vector prior to executing. ProbeMeister would modify the values in this vector by remotely invoking a method to alter the on/off values. However, (unlike method modification) object invocations using the JDI API require that the remote application be at a breakpoint. We have yet to measure the overall cost of this approach. Although, given that remote probe removal is on the order of 250 milliseconds it has yet to become a major issue.

While using the JDI API, we've noticed three important constraints. First, to modify a method ProbeMeister needs a copy of the complete bytecode of the class because critical pieces found in a .class file are not defined at the method level. This includes, for example, the bytecode boundaries in which a given attribute is valid, as well as the definition of exception handlers. Unfortunately, we have learned that the JVM cannot synthesize class definitions, so at this time ProbeMeister must have access to copies of all of the bytecode it may edit. Second, there is no straightforward method to reliably cause a breakpoint to occur in the remote JVM. While one can arbitrarily set a breakpoint using the JDI interface, the problem is knowing *where* to set the breakpoint. We have created a simple mechanism that allows ProbeMeister to cause a breakpoint at anytime (using our Breakpointer class as described earlier), but only if the application attaches to ProbeMeister at startup (because we know where the application will *begin* execution!). We

have not yet looked for a reliable way to port the Breakpointer class to the targeted application if ProbeMeister attaches to a running application. However, ProbeMeister needs to set breakpoints so it can invoke methods on remote objects.

The final constraint is that when an application connects to ProbeMeister there is no way to identify it. We have implemented a mechanism that will read special ProbeMeister-specific parameters that can be included in the command line (this requires ProbeMeister to invoke methods in the remote JVM to access these values). Preferably, such metadata would be made accessible via the JDI API prior to accepting a connection.

Another limitation is that our supplied probes cannot modify a method's arguments when the symbol table is not included in the class (a compile-time option can strip a class of its symbol table). However, a probe could modify these values by cross-referencing the original source, though we have not tried this. Not having the symbol table limits what a probe can do in a running application, for better or worse. Still, if needed, it is possible to access a method's local variables by statically instrumenting the source code. For example, we have instrumented the source code of some core JDK classes (e.g. java.io.File and java.net.URL) with special probes that provide access to more details than otherwise currently possible with our bytecode probes.

With respect to performance issues, we have noticed that while probe deployment is relatively quick, remotely deploying new classes appears quite costly – on the order of 100+ seconds. We have yet to investigate this issue to determine the source of the problem, but

we did notice significant bandwidth usage.

Like any other code writing, it is important to extensively test new probes as poorly written probes can easily cause catastrophic effects (the creation of the Statement Factory was intended to minimize such problems). And while the inclusion of exception handling in a probe addresses some of these concerns, it is still quite easy to write damaging code if one is not careful.

6 Plans

We are working to extend and enhance ProbeMeister. As mentioned earlier, probes need a distribution infrastructure to emit events. As the Java JDK 1.4 Logger can send logged data to a remote location, this will be a lightweight alternative to using Siena. If the application is already using this mechanism, then we could also merge and remotely route application output and probe output together. Furthermore, the Logger API defines logging levels that we plan to extend to control which probes emit events. We plan to explore this approach to turning on and off probes, in addition to the current "deploy, remove, and redeploy" approach.

Another feature we are exploring is to remove the limitation requiring local bytecode access so that a method can be modified, and probe installed. This requires that ProbeMeister have access to a copy of every classfile in which a probe might be deployed. To alleviate this, we plan to deploy helper classes into the remote JVMs that will load and transmit (back to ProbeMeister) the classfiles to be modified. This will also guarantee that the classfile used by the

application is the same version that ProbeMeister is modifying.

Currently, ProbeMeister is limited to blind instrumentation. That is, it does not display the source code, or allow the user to specify probe location as a source code line offset. We plan to extend our user interface to support the ability to specify the location of a probe similar to how breakpoints are placed within a debugger interface.

Finally, we plan to define some default probe configurations for addressing common monitoring needs, such as network activity, binding failures, and file access. This would allow a user to quickly isolate certain types of problems, after which they could manually deploy probes into specific components given what they had observed.

7 Acknowledgements

Many thanks to Sun Microsystems Java CAP team for providing access to, and support of, JDK 1.4 (special thanks to Jim Holmlund for his responsive support in debugging JDI-related issues). Thanks also to the reviewers for their invaluable feedback.

8 Related Work

This is a partial list of related Java-specific tools.

Bytecode Modifiers

- JOIE: The Java Object Instrumentation Environment , <http://www.cs.duke.edu/ari/joie/>
 - Geoff Cohen (Duke/IBM), Jeff Chase (Duke), and David Kaminsky (IBM), [Automatic Program Transformation with JOIE](#) in Proceedings of the [1998 USENIX Annual Technical Symposium](#)
- [CFParse](#) , <http://www.alphaworks.ibm.com/>
- [BIT: Bytecode Instrumenting Tool](#) , <http://www.cs.colorado.edu/~hanlee/BIT/index.html>
- [Jikes Bytecode Toolkit](#) , <http://www.alphaworks.ibm.com/tech/jikesbt>

- [Bytecode Engineering Library](http://jakarta.apache.org/bcel/) ,
http://jakarta.apache.org/bcel/

Commercial Probe Deployment Tools

- [JProbe Java Performance Tools](http://www.klgroup.com/jprobe/)
<http://www.klgroup.com/jprobe/>
- [JTrek](http://www.digital.com/java/download/jtrek/index.html) ,
http://www.digital.com/java/download/jtrek/index.html
- [NuMega DevPartner® Java™ Edition](http://numega.com) ,
http://numega.com
- RootCause -Java and C++ ,
http://www.ocsystems.com

Research Probe Deployment Tools

- NTWrappers - C++ -
http://www.teknowledge.com

Pre-instrumented JVMs

- [Jinsight](http://www.alphaworks.ibm.com/tech/jinsight) ,
http://www.alphaworks.ibm.com/tech/jinsight
- [eTective](http://www.averstar.com/products/etective.html) ,
http://www.averstar.com/products/etective.html
- [Binary Component Adaptation for Java](http://www.cs.ucsb.edu/oocsb/bca/index.html) (BCA),
http://www.cs.ucsb.edu/oocsb/bca/index.html

9 References

- [¹] Software Surveyor Project,
<http://www.objs.com/DASADA/index.html>
- [¹] DARPA DASADA Program,
<http://www.darpa.mil/ito/research/dasada/projects.html>
- [¹] D Wells and P Pazandak, “*Taming Cyber Incognito: Surveying Dynamic / Reconfigurable Software Landscapes*”, In *Proc of 1st Working Conference on Complex and Dynamic Systems Architectures*, Dec 12-14, 2001, Brisbane, Australia.
- [¹] Sun Microsystems JDK 1.4 Java Platform Debugger Architecture,
<http://java.sun.com/j2se/1.4/docs/guide/jpda/jdi/index.html>
- [¹] [A. Carzaniga](#), [D.S. Rosenblum](#), and [A.L. Wolf](#)
"Design and Evaluation of a Wide-Area Event Notification Service". *ACM Transactions on Computer Systems*, 19(3):332-383, Aug 2001
- [¹] Bytecode Engineering Library,
<http://jakarta.apache.org/bcel/>
- [¹] *M Coutinho, R Neches, et al, GeoWorlds: A Geographically Based Information System for Situation Understanding and Management*, In *Proc of 1st Intl Workshop on TeleGeoProcessing*, May 6-7, 1999, Lyon, France.

C. Software Specification Sheets

(C-1) OBJS ProbeMeister

(C-2) OBJS Gauge Tool Set

(C-3) OBJS EnviroProbes

(C-4) OBJS XML2Java

(C-1) OBJS ProbeMeister

ProbeMeister

Category(ies):	Probes/Gauges
Institution/Company:	Object Services and Consulting, Inc. (OBJS)
Description:	ProbeMeister is a tool for dynamically inserting into, and subsequently managing, probes in running Java programs. Both GUI and API interfaces are provided. Features are: • Dynamic code insertion - The application can be running when the changes are made. Changes can be made at any point during the execution. Changes go away when the application terminates. • Simultaneous Connections - ProbeMeister can monitor and instrument several applications simultaneously. • Distributed Insertion - ProbeMeister can connect to and modify remotely running applications. • Configuration Management - ProbeMeister can record all of the modifications, which can then be automatically re-applied at will ProbeMeister represents a complete redesign of OBJS' existing Java ByteCode Instrumentor (JBCI) which it replaces.
For more information:	Paul Pazandak - pazandak@objs.com
Assumptions:	Java 1.4
Status:	Research Prototype
Availability:	DASADA researchers may obtain download access by contacting Paul Pazandak
See also:	www.objs.com/DASADA/ProbeMeister.htm

(C-2) OBJS Gauge Tool Set

OBJS Software Surveyor Gauge Toolset

Category(ies):	Probes/Gauges (Dynamic Analysis & Tuning, Event Monitoring)
Institution/Company:	Object Services & Consulting, Inc. (www.objs.com)
Description:	Gauge ToolSet: The current set of gauges include Coalescer, EventMonitor, EventMerger, StackTracer, Historian, and Mapper. Coalescer merges streams of separately collected event information and renders this information on a timeline chart, performing limited aggregation of events by time interval. EventMonitor categorizes events by type and renders HTML- and XML-based displayable summaries with expandable detail. EventMonitor includes a web server to support browser-based access. It can be configured to subscribe to any subset of, or all, published events. EventMerger, an extension of EventMonitor, performs event unification prior to rendering. Event streams may report on the same activities, but at differing levels from within the application. EventMerger identifies related streams of events by analyzing event content (e.g. stack traces, event type/subtype, component names and other attribute values). This can help, for example, to view the overall activities of each probed component in the application. StackTracer converts streams of application events into a trace of program execution and emits an XML representation. The events emitted by a probe may be generated via several different execution paths involving the probed method. This gauge provides insight into frequency of invocation along each path. It can also be used to filter out paths (and therefore events) so that particular application behavior can be isolated for further analysis. Historian archives execution traces and computes statistics of behavior. Mapper provides a visualization of the time-based relationships between events of an application.
Formore information:	Contact Us. Paul Pazandak David Wells
Assumptions:	Target application can be written in any language. Version 1.0

	requires event dissemination via Siena. The Software Surveyor ToolSet v1.0 is implemented in Java 1.3 and has been tested under Windows 2000.
Status:	Active Research Prototype
Availability:	It is currently distributed as part of the Software SurveyorDemo v1.0 Distribution.
See also:	http://www.objs.com/DASADA/

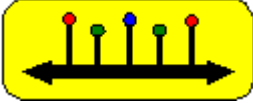
(C-3) OBJS EnviroProbes

OBJS Software Surveyor EnviroProbes

Category(ies):	Probes/Gauges
Institution/Company:	Object Services & Consulting, Inc. (www.objs.com)
Description:	EnviroProbes call upon operating system utilities together information on system status and resource usage. They monitor system-wide CPU utilization, application CPU utilization, and TCP bandwidth. They generate events at discrete configurable intervals.
Formore information:	Contact Us. David Wells Paul Pazandak
Assumptions:	EnviroProbes is currently available only on Win2000 and WinNT. Events generated use the Siena Event Dispatcher.
Status:	Active Research Prototype
Availability:	It is currently distributed as part of the Software Surveyor Demo v1.0 Distribution.
See also:	http://www.objs.com/DASADA/

(C-4) OBJS XML2Java

XML2Java

Category(ies):	General Support
Institution/Company:	 Object Services & Consulting, Inc.
Description:	Provides the ability to directly convert XML to first class Java objects (with application-specific behaviors). Potential use is to convert XML-based events to Java objects for application consumption.
For more information:	Contact Us. Paul Pazandak , David Wells
Assumptions:	Java 1.1+
Status:	Currently implemented in Lark XML Parser, but if there is interest we will embed it in the parser chosen for this project (as long as licensing permits and source code is available)
Availability:	TBD, see Status. Soon after a project parser has been chosen.
See also:	NA

D. User Manual

(D-1) ProbeMeister User Manual

ProbeMeister 2002 version 1.0/version 1.1

Copyright 2001-2003 Object Services and Consulting, Inc. All Rights Reserved.

See *OBJS_license.txt* for licensing. If file is not found, **do not use this software**. For licensing questions contact us at pazandak@objs.com or ford@objs.com.

www.objs.com

Thank you for your interest in **ProbeMeister**. **ProbeMeister** is well-documented in the sense that this paragraph contains a description of it, anything beyond that is a futuristic dream. **ProbeMeister** enables the insertion of new code (**probes**) into running distributed Java applications. It has a number of predefined probes that can be used. Once a probe is inserted into a method of a class, it will be executed the next time the method is invoked. Probes can also be removed.

Probably the best way to understand **ProbeMeister** is to try it out. Run the demo below...now! A paper is also available on ProbeMeister on request.

Running the demo

A demo is included. The primary thing to note is the .bat file so you can understand how a "target VM" is attached to **ProbeMeister**. The demo is started by invoking the *runSimpleExampleClient.bat*. The main of SimpleExample2 calls printPing2(String, String, int) -- so this is the method you should instrument if you want to see something happen.

Getting Started:

1. The first thing to do is **start ProbeMeister**. The example will not run if **ProbeMeister** is not running. There are ways around this, but then also caveats. See the **Help...** under **Virtual Machines** menu to get more on this.
2. Now, start the demo - **invoke runSimpleExampleClient.bat**
3. The first thing to notice is that "Add Available VM" button is now enabled. **Click "Add Available VM"** to have **ProbeMeister** attach to the JVM (*target VM*) running the example.
4. All of the initial classes of the target VM are displayed, up to but not including the main class. Then the target VM is halted by **ProbeMeister**. This break allows

you to instrument core Java classes before the application starts. Hit **Resume button** to load the remaining classes.

5. At this point **ProbeMeister** will load the main, do some stuff including reading any command line identifiers so we know what application we've attached to (yes, this is a 'defect' of the current JDK API, but hopefully it will improve in the next release). **Wait 5-10 seconds** for this to happen.
6. Now, you can see all of the loaded classes. Use the filtering options (e.g. "**Exclude JDK classes**") at the right in the **ProbeMeister** GUI to filter the list of displayed classes. One probe has already been installed (if all went well) in case you were wondering how it got there. If you see less than four classes, hit the filter button again, it will refresh the view. (**Note**: you cannot instrument inner classes at this time - their names include a '\$').
7. The example should have opened it's own GUI, so you should try it out & see what is displayed in the console before you insert any probes.

Probe Insertion:

1. **Select the main class SimpleExample2**, it should expand to display its methods.
2. Select instrumentMe() method. **Drag a method from the list on top of this method.** The simplest of these is **PrintString**. As it suggests, it simply prints a user supplied string in the console of the target VM.
3. **Select PrintString probe. Enter a string to print in the dialog (e.g. "Got here..."). Hit OK.** A new entry for this probe should now appear under this method in the list. Select the new entry to see a description of this probe.
4. Now, to see if it worked, bring the SimpleExample2 gui and console to the front, then **press a radio button**. You should see your string printed to the console of SimpleExample2.
5. To remove a probe, right-click it and select "**Delete Probe**" from the pop-up menu.

That's it. There are two types of probes:

- **Simple probes.** They are self-contained, including all the code they need to execute.
- **Probe Stubs & Plugs.** Stubs are not self-contained. They make a call out to another class/method (a plug). The plug contains the body of the code to be executed. This is particularly useful if the code is used by several types of probes, or if it is so complex or long that you don't want to have to define it all in bytecode! While simple probes and probe stubs **must** be defined in bytecode, plugs are just normal methods in compiled java classes.

Stubs are paired up with plugs based upon their signatures. Stubs and plugs define their signatures, and **ProbeMeister** automatically finds appropriately matching plugs, and asks the user to select one (there may only be one choice for some stubs).

Stubs and simple probes need to be registered in the ProbeCatalog.txt (located in the jar file). Probe plugs must be registered in **ProbePlugCatalogDB.txt**. You may add new plugs easily. See the example source file (**PLUG_GeoWorldsEventsToSiena.java**) in the libraries directory. It shows how one can easily write probe plugs (there is a lot of superfluous code -- it's quite easy to write a plug). All one must do is to create an entry in the catalog, and then create methods that start with "**PP_**" in the class. The methods must match at least one signature of a probe stub, otherwise it cannot be called by a stub.

Probe Types

Here's a description of the available probes.

- **PrintString** - prints a user-supplied string in the target VM console
- **CallMethod** - calls the specified STATIC method in the remote VM. No exception handling, so the method may throw an exception if an exception occurs in the static method called, or if an exception occurs in trying to call the method -- e.g. it wasn't really a static method. Use "Validate Method" in the probe dialog box (displayed when adding the probe) -- it'll make sure that the method exists & is static.
- **Stub_CallMethod** - Calls a probe plug having a simple no argument list. By default it calls a simple plug that emits a string describing the method that called it.
- **PassMethodArgsStub** - Like the above stub, but (should) pass the methods arguments to the plug to be printed. May need some work.
- **BasicEventStub** - Emits an event to be consumed by the Siena Distributed Event Server. You need to have Siena server running. To monitor the emitted events (remember they are only emitted when the probe code is invoked), run the event monitor which brings up a browser window. It autorefreshes about every 10 seconds. The emitted events are categorized by event name - a name you provide. The stub actually calls a plug that emits the event.
- **PassMethodArgsEventStub** - Like the above stub but it passes all of the instrumented method's arguments within the emitted event. Obviously, if an argument is an object it will be passed as a representation of that object.
- **PassObjectEventStub (new)** - Like the above, but it passes 'this' of the current method. It will break if the method you instrument is a static method I would guess. The receiving plug, if customized to manipulate a given object type, can extract all sorts of information from the object. See **PLUG_GeoWorldsEventsToSiena.java** which has a plug that accepts a generic object, and extracts data only if it's a GeoWorlds ServiceProxy object.
- **CallMethodByName** - Unlike the **CallMethod** probe, this probe calls by Java's introspection wrapped with exception handling. So, if the specified method doesn't exist, the called method throws an exception, or even if the invocation of the specified method causes an exception, the exception will be handled (exception information will be sent to the console of the target VM should an exception occur). The benefit is that the invocation of the probe will not have any adverse effect on the execution of the instrumented method.

Other Notes

Connections:

To remove an application, select "Disconnect" and then "Remove". The remote application remains running, and any probes deployed will remain until the application terminates. If the application was started as a server (see the Virtual Machines help menu in ProbeMeister), then you can always reconnect at a later time & modify the probes that you deployed.

Note that you can add several applications to ProbeMeister!!

Probe Configurations:

See the Configurations help menu in ProbeMeister.

Remote Control ala RMI

Everyone likes remote controls, so we added one. See the runTestRemoteRMI.bat and TestRemoteRMI.java files. This capability allows a remote application to deploy probes, or to suggest locations for a probe to be deployed via a Gauge Deployment Request. More information on request!

Notes

Send questions to me: pazandak@objjs.com

ProbeMeister outputs general messages to the console. It also outputs detailed messages & errors in XML in diag.pml. Send this file to me if you need help debugging a problem.

E. Demonstrations & Tech Transfer

(E-1) IntelliGauge TIE - Using Gauges Throughout the Software Lifecycle to Improve Internet Information Systems, IntelliGauge Project Team, October 2000

Abstract: Description of a group effort to apply a suite of DASADA technologies to monitoring, diagnosing, and tuning a loosely coupled Internet-based application.

IntelliGauge TIE

Using Gauges Throughout the Software Lifecycle to
Improve Internet Information Systems

Year 1 Group Plan

October 2, 2000

Participants

BBN Technologies

Columbia University

Object Services

USC Information Sciences Institute

University of Colorado - Boulder

Veridian

WPI

Hypothesis

Software gauges can:

- Efficiently and transparently monitor distributed, real-world software to collect, analyze, and disseminate information to Solve configuration and usage problems at all points in the software life cycle

Approach

- Demonstrate how DASADA gauges can non-invasively instrument a complex real-world software application (GeoWorlds) that is typical of Internet-based intelligence gathering, analysis, and planning systems.
- Demonstrate how DASADA gauges can be used to diagnose and assist in the repair of composition and operational problems throughout the software life-cycle.
- Demonstrate the effectiveness of the gauges by using them to diagnose real configuration and operational problems as reported by existing GeoWorlds users.

Technology Sharing Plan

There is a potential for overly tight interaction between groups, so we agreed to limit dependencies to:

- Agreement on key definitions
 - events and event posets defined by a FleXML (meta-)Schema
 - probes & gauges specified in Acme
- Common infrastructure
 - Sienna as common event distribution mechanism
 - sharing of (but not reliance on) individual probe & gauge placement tools
- Loose (first year) coupling between different projects' probes & gauges
 - limited first year consumption of other project's probe & gauge inputs/outputs
- Common demonstration application with individual "mini-demos" in scenario-based framework

Producer/Consumer Relationships

TBASSCO (USC/ISI) produces

- Semantic service and data flow description capability
 - BBN, Veridian and Object Services use them to describe semantic interoperability of their services
- Service event protocol specification based on semantic service description
 - Columbia/WPI verifies services are *conforming* during runtime

TBASSCO (USC/ISI) consumes

- Semantic distance metric to measure interoperability
 - Georgia provides metrics, i.e., clustering and factor analysis
- Runtime performance to tune architecture
 - Columbia/WPI, Object Services, BBN provide performance gauges
- Runtime service quality to select alternative services
 - Columbia/WPI provides quality gauges, i.e., size of search result

- Veridian/PSR Produces
 - A callable web service for creating GIS products for GeoWorlds
 - An update to the Venice application framework for dynamically (re-)configuring this web service
- Veridian/PSR Consumes
 - Nothing in first year

- BBN Produces
 - “Abstract Query Engine” “applet/agent” for demonstration with GeoWorlds
 - Website wrappability gauges
 - Runtime quality assurance content-level gauge
 - XML Binding Adapter. Plug'n play XML technologies to dynamically update/ manipulate ADL XML.
- BBN co-Produces
 - Service Contract Language
- BBN Consumes
 - Event Language (external interface to Gauge Infrastructure)
 - GeoWorlds infrastructure
 - Other performance gauges (BBN will be producing 2).

- **Columbia/WPI Produces**
 - AIDE toolkit for inserting active interface probes into Java code.
 - FleXML toolkit - including Schema templates for defining event vocabularies, Oracle for publishing new vocabularies, Metaparser for validating and preprocessing event streams, converter to/from Siena.
 - Worklets toolkit for deploying/modifying live probes & gauges, emitting and coordinating dynamic reconfiguration gaugents.
 - Sample probes & gauges for monitoring GeoWorlds protocol compliance.
- **Columbia/WPI Consumes**
 - ISI GeoWorlds infrastructure and protocol specs for main demo.
 - OBJS Smart Data Channels for (optional) PDA demo.
 - UColorado Siena for transporting FleXML event streams.
 - CMU xAcme activity language to ensure FleXML compliance.
 - UMass Little-JIL decentralized workflow for worklet oversight.

- **University of Colorado Produces**
 - FIRM probe and gauge deployment infrastructure, which includes installation, activation, and deactivation
 - Siena wide-area event notification service
 - Sample probes & gauges for monitoring proper deployment of GeoWorlds components
- **University of Colorado Consumes**
 - ISI GeoWorlds infrastructure and protocol specs for main demo
 - UCI and CMU xADL joint architecture description language
 - FleXML toolkit - including Schema templates for defining event vocabularies, Oracle for publishing new vocabularies, Metaparser for validating and preprocessing event streams, converter to/from Siena

Object Services (Software Surveyor) will produce the following software that will be used by others:

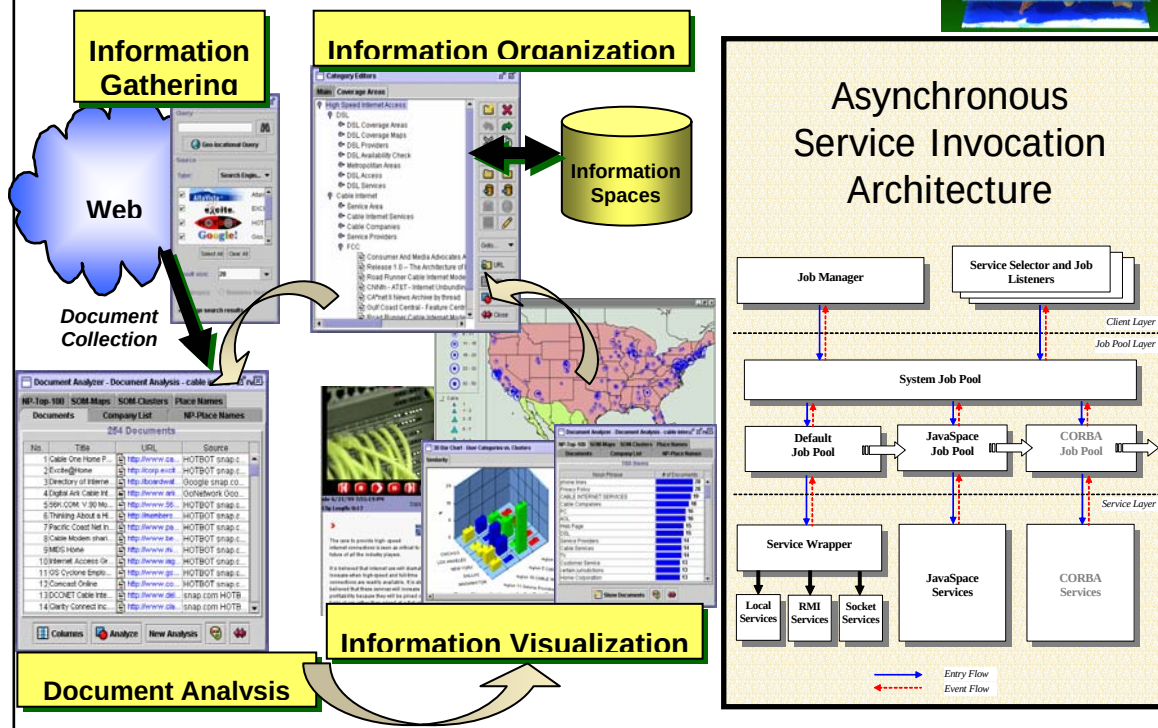
- System Administrators use *ConfigMapper*, *ConfigComparator* & *ConfigChecker* to diagnose GeoWorlds installation and reconfiguration problems.
- TBASSCO (USC/ISI) use *ConfigComparator* and *ConfigChecker* to help end user Intelligence Analysts to diagnose the sources of suspicious query results and identify inconsistencies in query construction.
- BBN and Columbia/WPI use *ConfigMapper* to determine where activity monitoring and QoS probes should be installed.
- The Event Infrastructure may use *XML2Java* to map XML-encoded events to Java-encoded events.
- Gauge Developers may use *JBCI* to place probes and stubs into applications.

Software Surveyor consumes the Event Dissemination Infrastructure & GeoWorlds demo.

Demo Structure

- Illustrate gauge use in several “problem/diagnosis/response” scenarios in 4 distinct GeoWorlds lifecycle activities
 - Deploying/Installing GeoWorlds
 - Information Management Scripting
 - Script Execution
 - Reconfiguring GeoWorlds
- Common demo theme across projects
 - Common storyboard across the lifecycle
 - Each scenario within a lifecycle activity shows one project’s capabilities
 - Individual scenarios will be grouped to show a combined capability
- Equipment assumptions
 - LAN or wireless connectivity + T1 Internet access

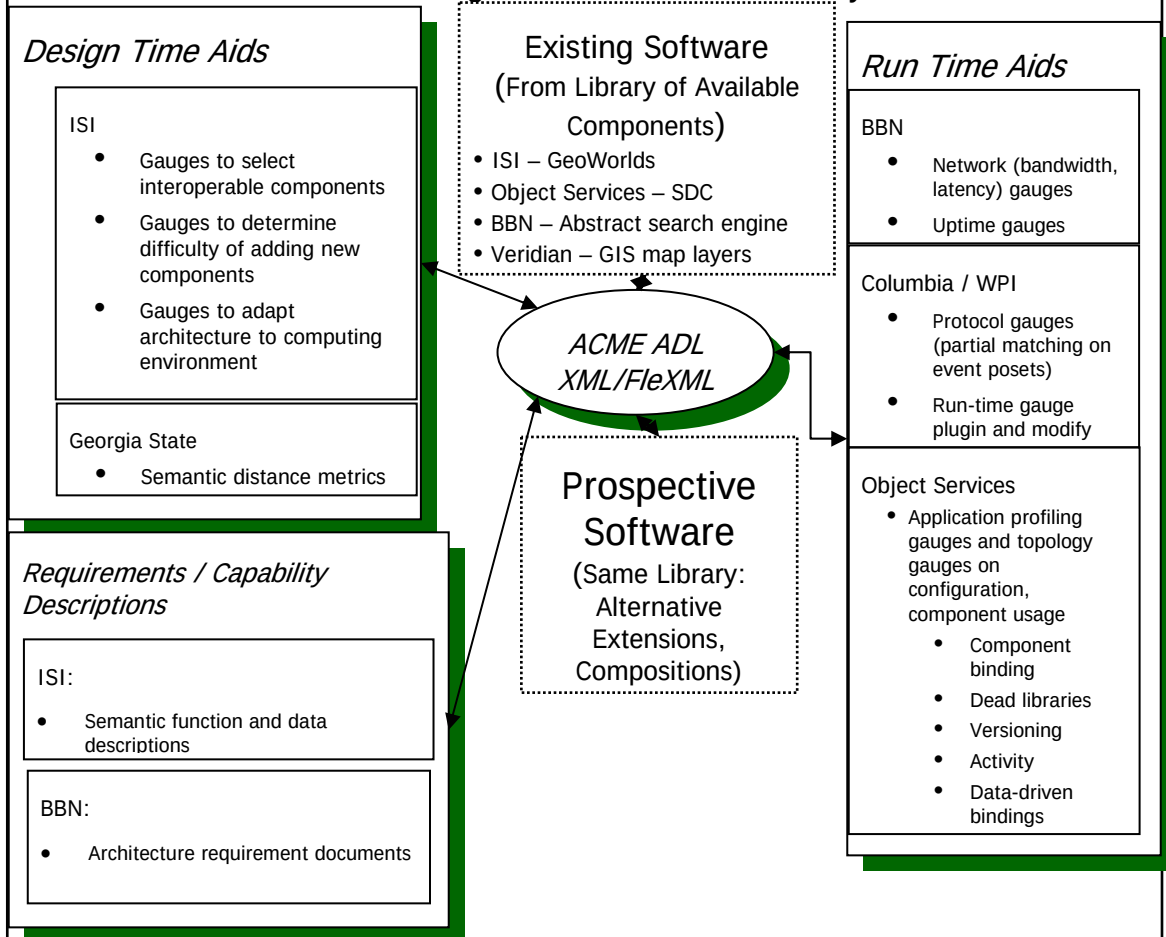
GeoWorlds Test-bed Application



GeoWorlds Test-bed Application

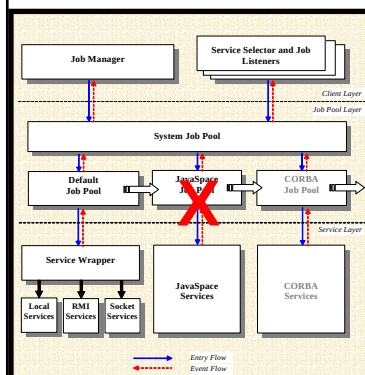
- Large component-based system in use at PACOM
 - PACOM and JFCOM are potential outside evaluators
- Framework for adding components
- Geographic Information Systems plus Web processing
- Ops and intelligence uses, e.g.,
 - Mapping terrorist bombings
 - Locating recurring natural disasters
 - Investigating drug trafficking and piracy in various locales

Functionality to be Illustrated: Probes & Gauges in the Software Lifecycle



Lifecycle Scenarios:

1. Installation Time



User Observes	Causes	Gauges Do	How
Install script fails	Expected component not found	Config gauge identifies missing component	Compare installed config w/ Acme spec
Installation completes, but GeoWorlds doesn't work	Version mismatch	Config gauge identifies use of different version	Compare installed config w/ a good installation
	Namespace error		
	Method invocation fails		

2. Information Management Scripting Time

Probes & Gauges

User Observes	Causes	Gauges Do	How
GeoWorld script cannot be completed	I/O data semantic mismatch between components	Semantic gauges identify the mismatch & suggest intermediate components to resolve it	Perform reasoning on I/O semantics and find components that make a semantic connection
	Syntactic (interface) non-compliance between components	Syntactic gauges determine the cause of non-compliance and suggest adapters	Access to library of converters and wrappers
	Dataflow violation (e.g., pipe output, page input)	Dataflow gauge detects mismatch & suggests a dataflow adaptor to allow	Access to library of available converters
	GeoWorlds can't find appropriate data source	Semantic gauge subscribes data channels that meet requirement	Compare the data requirement and channel descriptions

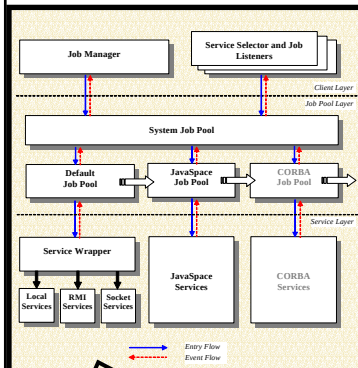
Intelligence Analyst

3. Script Execution Time

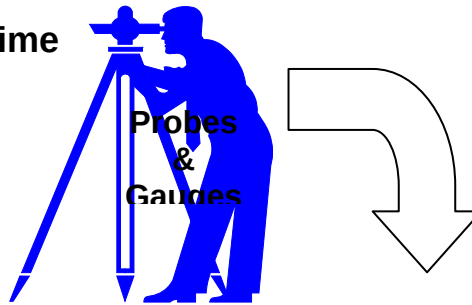
Probes & Gauges

User Observes	Causes	Gauges Do	How
GeoWorlds hangs during script execution	External data source has moved, disappeared, or become unresponsive	Connectivity gauge identifies the broken connection & finds new URL or drops the source	Monitoring request & response pairs and comparing timing with previous interactions
	External data source has changed its (XML) interface	Change monitoring gauge determines an XML encoding has changed	Comparing the XML Schema used in a sequence of accesses
	External service failure causes GeoWorlds script failure	Connectivity gauge identifies broken connection & suggests alternate service	Monitoring request & response pairs and comparing timing with previous interactions + knowledge of service alternates with similar interfaces
Script returns suspicious results	Spam site introduces flood of dubious responses	QoS gauge identifies aberrant site & helps build a filter to eliminate spurious results	QoS gauge identifies growth in result size Dataflow gauge identifies the path through the query that caused the increase

4. Reconfiguration Time



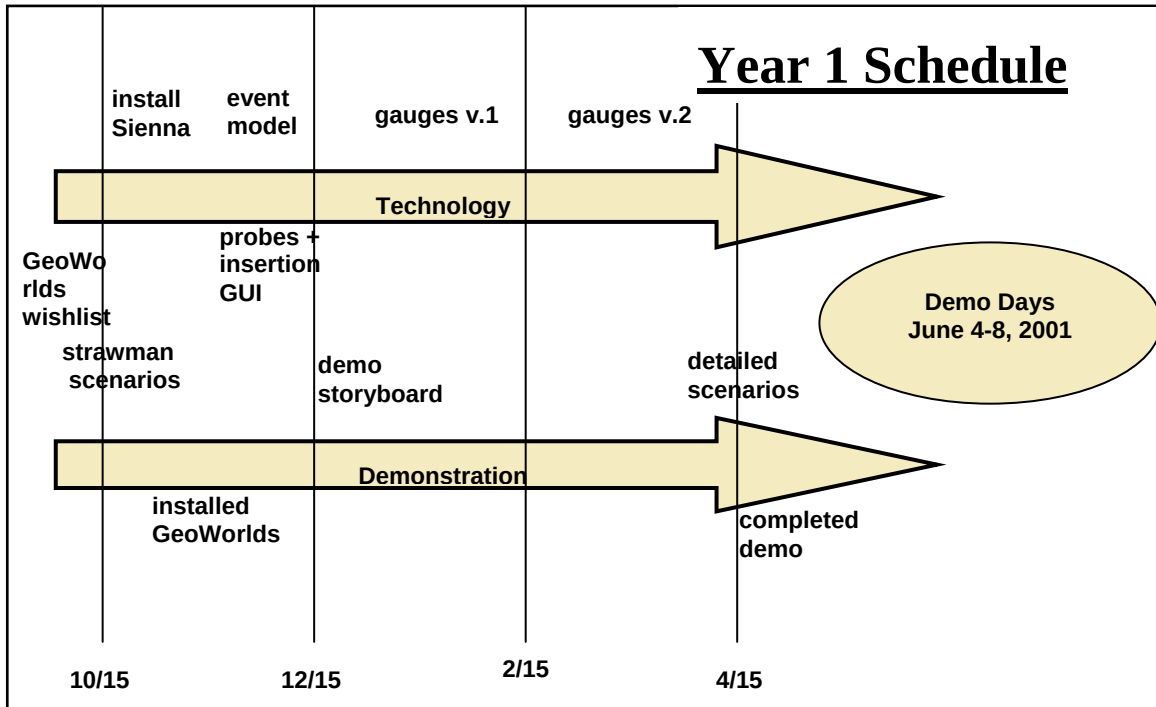
Brilliant
new
service



User Observes	Causes	Gauges Do	How
Attempt to install new remote service fails	Should not happen - GeoWorlds will accept any service		
Users don't know how difficult it is to add a new service	Mismatch with GeoWorlds data and service specifications	Conformity gauges measure how well do I/O data and functionality conform to GeoWorlds data and function specifications	Check conformity to document models and service APIs, and data and functional ontologies. Check if any wrapper can be applied
Installation Completes, but GeoWorlds doesn't work with new service	GIS Data mismatch with request	Semantic matching of components in reconfiguring the service	Venice allows the user to reconfigure the service remotely using an ACME description for the architecture

Group Evaluation Criteria

- How *efficiently* GeoWorlds can be installed in different environments and its services deployed.
- How *easily* complex information management tasks can be scripted with assured semantic and syntactic interoperability.
- How *reliably* the scripts can be executed while maintaining desired quality level.
- How *dynamically* the scripts can be evolved based on resource availability and requirement changes.
- How *efficiently* can new services be added to GeoWorlds while maintaining compatibility.



Coordination Mechanisms

- Source code using Source Forge technology
- Effective network of web-based sharing of documents
 - BSCW hosted by Columbia
- Develop Architecture for entire system showing group involvement
- Conference calls and email
- ICSE 2001 in Toronto
- Winter PI Meeting
- Face-to-face meetings
 - Individual sub-groups only
- Working Demo by May 1st

Outside Interactions

- Interaction with other DASADA groups
 - Eliminate redundancy
 - Propagate developed standards and standards in progress
 - Produce schedule for our deliverables

Event “wire format” and dissemination mechanisms	University of Colorado, Teknowledge
ACME representations and tool kits	Carnegie Mellon, University of California-Irvine
Probe toolkits or infrastructures	Teknowledge
Gauge toolkits or infrastructures	Multiple groups

(E-2) Software Surveyor: Dynamically Mapping Untamed Software Applications, OBJS Project Brochure, June 2001

Abstract: Project overview and description of the status of the tools as of mid-2001.

Software Surveyor

Mapping Untamed Software Applications



Object Services and Consulting, Inc.



Cyber Incognita

Terra incognita – the unknown territory that baffled explorers, frightened merchants and impeded progress. Difficult to know where you are and impossible to know what to expect next. *Hic sunt dracones* – here lurk dragons.

The power and flexibility of modern software makes it increasingly a *cyber incognita* and the traditional tools of maps (design specs), surveying and navigational instruments, and marked trails (descriptions of normative behavior) are as inadequate in *cyber incognita* as they were in *terra incognita* 300 years ago. Design specs are incomplete, inaccurate, or inconsistent; software probes cannot observe all significant events and

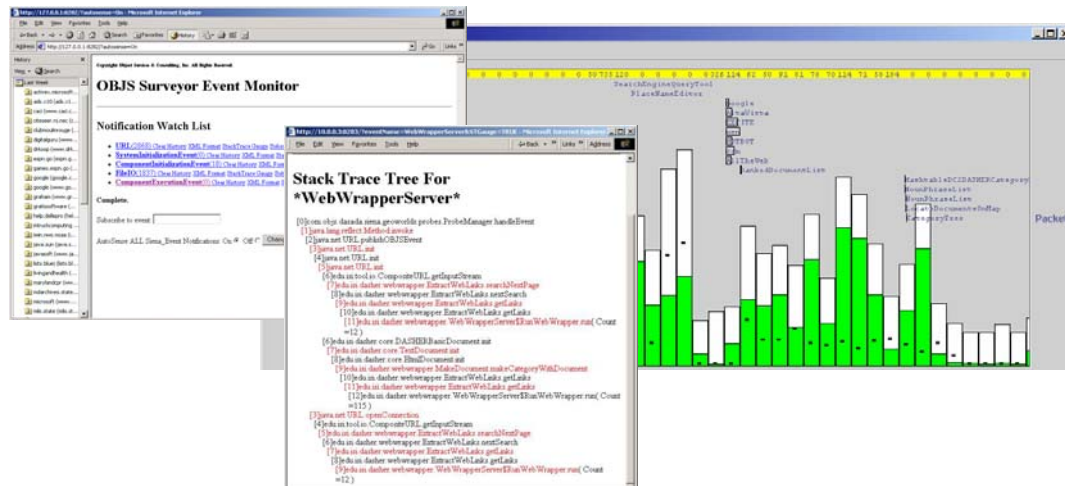
techniques to correlate independently recorded observations are limited; and descriptions of normative behavior are often (especially in Web-based, agent, or survivable systems) described as “best effort” with no concrete notion of what that means. Further, the dynamic nature of many modern applications means that they are continually reorganizing themselves in response to changed user demands or resource availability; the equivalent of Lewis and Clarke having to deal with rivers and mountains that changed position every few hours.

So, if you have ever felt that using and managing complex, distributed (and often under-specified) dynamically reconfigurable software is a bit like walking alone into the wilderness, *Software Surveyor* is for you.

Software Surveyor Overview

OBJS’ **Software Surveyor** is a profiling toolkit to dynamically deduce and render the runtime configuration and behavior of evolving, component-based software. Infor-

mation is synthesized from multiple sources and combined and rendered in a variety of formats and made easily accessible via the Web.



Software Surveyor addresses three distinct issues:

- What is the application doing?
- What is it supposed to be doing?
- Is it doing what it is supposed to?

This requires probes to collect a variety of information and an infrastructure to disseminate it, and synthesis tools to merge information streams and make sense of it. Results of this analysis are aggregated to iden-

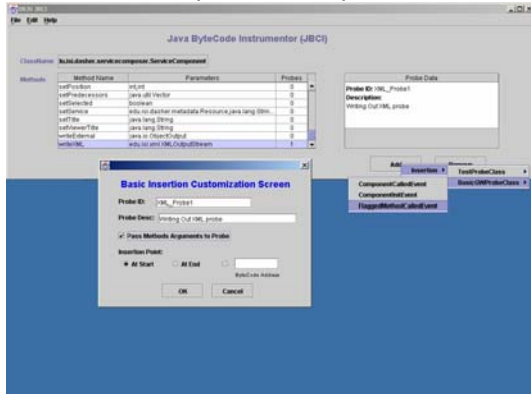
tify "behavioral norms" to augment incomplete performance specifications. Finally, the probe infrastructure and behavioral norms can be used to signal users when the system is operating anomalously. *Software Surveyor* requires limited prior knowledge of application connectivity and has the ability to dynamically deploy probes, allowing its use with dynamically reorganizing applications and those lacking complete specifications.

Current Tools

Probes

AppliProbes provide information about events at the application interface and/or internal to the application. EnviroProbes uses operating system utilities to gather and emit information on system status and resource use.

The Java ByteCode Instrumentor automates the insertion of probes and probe stubs into



Java ByteCode. JBCI modifies .class files by inserting calls to selected probes using selected customizable instrumentation techniques. JBCI can be extended with new probes and instrumentation techniques. GUI and programmatic interfaces will be available. Probes implemented in other languages can be called via JNI. The next version of JBCI will support on-the-fly probe insertion into running programs.

Event Distribution

Events are distributed by the Siena Event Distribution Infrastructure (U-Colorado). XML2Java translates XML to first class Java objects with application-specific behaviors. Useful to convert XML-encoded events into a readily manipulable form.

Analysis

Coalescer merges streams of separately collected event information to create an event timeline and performs limited aggregation of events by time interval. StackTracer converts streams of application events into a trace of program execution and emits an XML representation. EventMonitor categorizes events by type and produces summaries with expandable detail. Historian archives execution traces and computes statistics of behavior.

Visualization

Mapper provides a timeline-oriented visualization of application behavior. StackTracer and EventMonitor results can be viewed using any XML-capable Browser.

Implementation & Status

Software Surveyor v1.0 is implemented in Java 1.3 and has been tested under Windows 2000. v1.0 requires Siena for event distribution. EnviroProbes is currently available only on Win2000 and WinNT.

4.4.1 Principal Investigator: David Wells

4.4.2 Investigator: Paul Pazandak

For more information:

www.objs.com/DASADA

{wells, pazandak}@objs.com

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government .

(E-3) Software Surveyor: Dynamically Mapping Untamed Software Applications, OBJS Project Brochure, June 2002

Abstract: Project overview and description of the status of the tools as of mid-2002.

Software Surveyor

Dynamically Mapping Untamed

Object Services and Consulting, Inc.



Cyber Incognita

Terra incognita – the unknown territory that baffled explorers, frightened merchants and impeded progress. Difficult to know where you are or what to expect next. *Hic sunt dracones* – here lurk dragons.

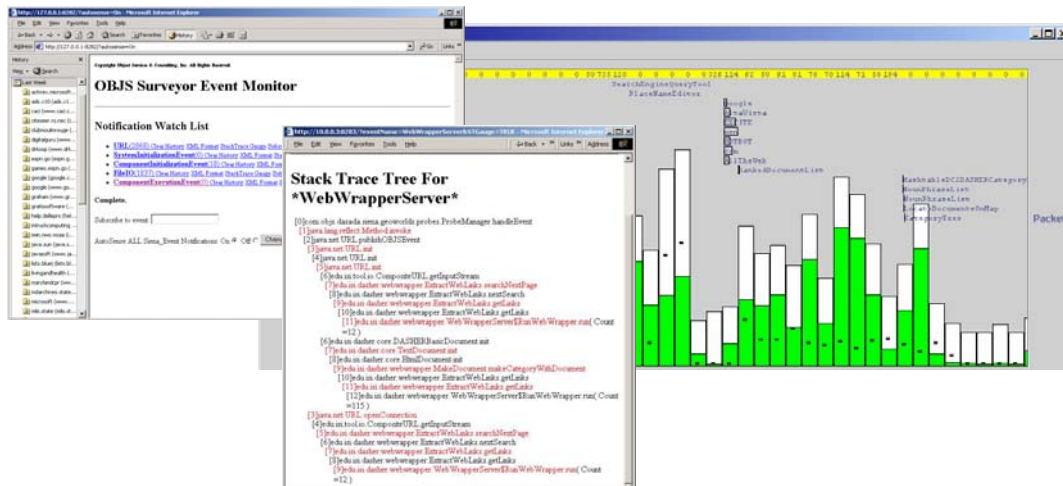
The power and flexibility of modern software makes it increasingly a *cyber incognita* and the traditional tools of design specs, monitoring software, and descriptions of normative behavior are as inadequate in *cyber incognita* as maps, marked trails, and surveying equipment they were in *terra incognita* 300 years ago. Design specs are incomplete, inaccurate, or inconsistent; software probes cannot observe all significant events; techniques to

correlate independently recorded observations are limited; and descriptions of normative behavior are often (especially in Web-based, agent, or survivable systems) described as “best effort” with no concrete notion of what that means. Further, the dynamic nature of many modern applications means that they are continually reorganizing themselves in response to changed user demands or resource availability; equivalent to Lewis and Clarke having to deal with rivers and mountains that changed position every few hours.

So, if you have ever felt that using and managing complex, distributed (and often under-specified), dynamically reconfigurable software is a bit like walking alone into the wilderness, *Software Surveyor* is for you.

Software Surveyor Overview

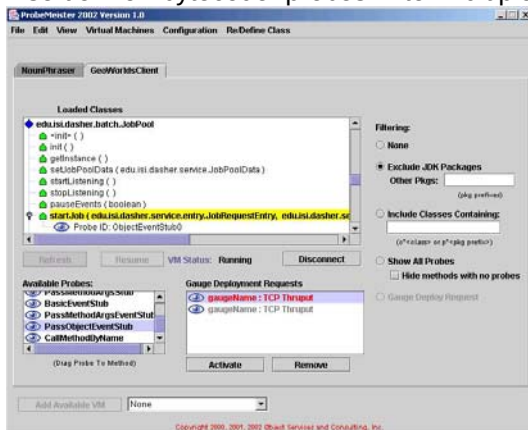
OBJS’ ***Software Surveyor*** is a profiling toolkit to dynamically deduce and render the runtime configuration and behavior of evolving, component-based software. Information is synthesized from multiple sources, combined and rendered in a variety of formats, and made easily accessible via the Web. *Software Surveyor* requires limited prior knowledge of application connectivity and has the ability to dynamically deploy probes into distributed applications, allowing its use with dynamically reorganizing applications and those lacking complete specifications.



Current Tools

Probes

Software Surveyor's core technology is *ProbeMeister*, a dynamic probe deployment tool. ProbeMeister manages the dynamic insertion of bytecode probes into multiple



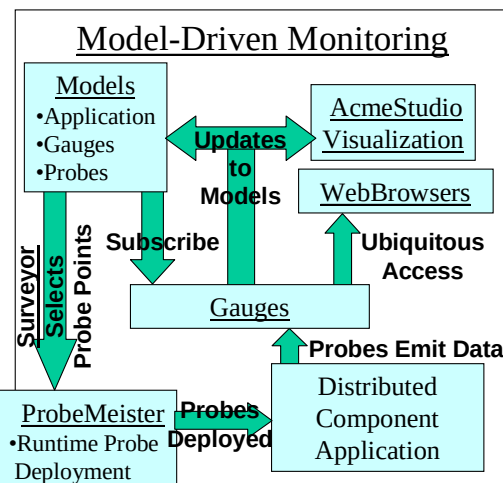
running distributed Java applications. ProbeMeister modifies applications at run-time by inserting customizable bytecode probes into selected methods. ProbeMeister can be extended with new probes and instrumentation techniques.

ProbeMeister provides a remote method interface (RMI) enabling external applications to control probe deployment. Probes implemented in other languages can be called via JNI.

ProbeMeister also provides configuration management, so frequently used probe configurations can be saved & reused.

Model-Driven Probe Deployment

Controlling ProbeMeister is our **model-driven Gauge Management** tool. Software Surveyor's Gauge Manager uses an application's architectural specification model to iteratively determine where to deploy probes. This helps to simplify the task of knowing where to probe the application to monitor what it is doing.



Analysis

Once the Gauge Manager has deployed probes, Software Surveyor's Gauge Analysis Tools consume and interpret the data emitted by the probes. Gauges have been built to construct stack traces of remote applications, merge environmental and application-specific information from multiple probes, and organize and archive observed events.

Visualization

Mapper provides a timeline-oriented visualization of application behavior. *StackTracer* and *EventMonitor* results can be viewed using any XML-capable Browser. Gauges can be attached to applications viewed in the AcmeStudio architectural visualization tool, which can also be used to initiate gauge selection and deployment.

Implementation & Status

Software Surveyor v2.0 is implemented in Java 1.4 and has been tested under Windows 2000. v2.0 requires Siena for event distribution. *Software Surveyor* is compliant with the evolving DASADA gauge and probe infrastructures.

4.5.1 Principal Investigator: David Wells 4.5.2 Investigator: Paul Pazandak Object Services and Consulting, Inc. USA	For more information: www.objs.com/DASADA {wells, pazandak}@objs.com
This work is funded by the Defense Advanced Research Projects Agency and monitored by the Air Force Research Laboratory F30602-00-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government .	

(E-4) Demo Abstract, OBJS Report, June 2002

Abstract: A short description of the 2002 Software Surveyor Demonstration applying Software Surveyor to the GeoWorlds intelligence gathering system described in the IntelliGauge TIE paper. OBJS Software Surveyor

Demo Abstract

June 2002

Against a backdrop of the of the IntelliGauge TIE, in which a coherent collection of DASADA technology is used to model, tailor, monitor, analyze, and repair a geospatial situation understanding system (GeoWorlds) used at US Pacific Command (PACOM), OBJS will demonstrate: 1) use of architecture models to determine how/where to probe target applications, 2) the ProbeMeister Probe Manager for runtime insertion and management of probes in remote and running Java programs, 3) use of the Siena wide area event distribution bus, 4) a collection of gauges to collate and interpret sensed events, some of which are architecture-model sensitive, and 5) event logging to support off-line analysis and for use by third-party repair tools.

(E-5) White Paper – Dynamic Modeling and Analysis Tools for Cougaar

OBJS White Paper to the DARPA UltraLog Program, 2002, Mark Greaves Program Manager.

Abstract: Description of how Software Surveyor could be used to monitor & debug distributed agent systems in the DARPA UltraLog Program.

White Paper: Dynamic Monitoring and Analysis Tools for Cougaar

David Wells & Paul Pazandak

Object Services and Consulting, Inc.

{wells, [pazandak](mailto:pazandak@objs.com)}@objs.com

© Copyright 2002 Object Services and Consulting, Inc. All Rights reserved.

Problem: Both the Cougaar infrastructure and application societies built on it are so complicated and dynamic that it is difficult to detect or diagnose problems. Monitoring and modeling are necessary throughout the lifecycle for development/debugging, deployment, health monitoring, and performance tuning/improvement.

Remote, distributed debugging is needed during development. Many processes send messages to other processes, but a single debugging environment only lets you see what's happening "here" – the process that is sending the message. There is no way to know if the message was received, if the right recipient got it, what/if any action is being taken by the recipient or whether the recipient has died or been delayed.

The configuration of a deployed Cougaar society (at both the application & infrastructure levels) is the result of a complex interaction between many different kinds of information created by one or more independent actors (e.g., software engineers, administrators, logisticians, enemies) and representing a combination of application-level objectives, infrastructure-level actions, and outside events. A configuration is established by combining static specifications (e.g., available plug-ins & business rules), deployment actions (e.g., choice of business rules & connections to local defaults), runtime decisions (e.g., the result of a yellow pages lookup), and unintentional actions (e.g., attacks and failures). This interaction can be quite complex as the following example shows. Assume that an infantry company is configured to order water from only reasonable places. The command structure dictates that deliveries must be made by trucks belonging to brigade support. Localization rules dictate that commodities like water must be bought locally and if possible from a host nation provider. Purchasing rules dictate that everything must be bought from a supplier on a pre-approved list. When the infantry company deploys, these rules are combined to determine the set of legitimate water suppliers and shippers. But what if the rules are wrong or in conflict? What if the brigade trucks did not deploy and are 5000 miles away? Can a plan be made to ship local water using such trucks? What if a failure causes the only allowable supplier to stop responding? What if one of the suppliers on the list is captured by the enemy and is no longer acceptable? In each of these cases, it is necessary to identify the set of contacts deemed feasible *in the particular deployment*, update this dynamically as the situation evolves, present it in a way that a logistician can determine if the result makes sense, and track connectivity problems back to their source in the rules/defaults/trades/etc.

Health monitoring is required input for runtime survivability action. Foremost for ensuring robustness is knowing whether something has failed, if a response is delayed, if there are unexpected loads, etc. Security monitoring includes detecting with whom agents/nodes are actually communicating, including side channels as well as through the approved Cougaar mechanisms. If there is a suspicion that a component has been compromised, can it be monitored reliably and without letting an attacker know that it is being monitored? It is also useful to have sufficient information to allow unsuccessful executions to be compared with successful ones to identify differences (e.g., new plug-in, change node connectivity or usage, node failure, or corruption of a node) that might have caused the problem.

Both survivability and performance tuning require information about activity patterns and available resources as inputs to prioritization and resource re-allocation mechanisms. Performance tuning/improvement requires knowing what's been happening to determine where the most bang for buck is available. For instance Cougaar societies will be very sensitive to tasking fanout; if a complex task is not distributed to multiple nodes, insufficient computing power at a single node will be a bottleneck; on the other hand, if tasks have too great a fanout, communications will become a bottleneck.

Desirable Characteristics of a Monitoring System: Support throughout multiple phases of the software lifecycle and the ability to serve many different monitoring needs under the same umbrella is clearly desirable. This will require many kinds of probes to collect execution information about applications and the Cougaar infrastructure and resource utilization information at both the O/S and network levels. This includes both information from within the normal Cougaar framework, but also potential side-channels by which rogue agents could be divulging information.

Since applications are distributed, this information must be able to be combined remotely. This implies a common *event infrastructure* through which detected events are collected and a common *event schema* which will allow separately collected information to be interpreted in a common form. Events must be correlated to be useful. While it is easy to order events observed in a single environment, this is of limited use in Cougaar where many activities span machine (and hence clocking) boundaries. A partial ordering between events observed across environments (and hence subject to clock skew) can be created using additional sequencing information such as the knowledge that a message send in environment A precedes a message receive in environment B, thereby allowing a correlation between event streams.

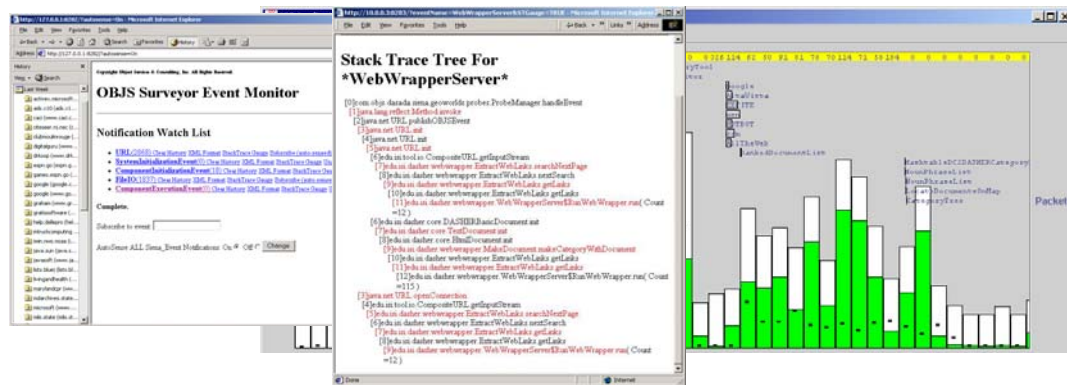
Finally, it is highly desirable to be able to add monitoring to a running application without having to restart it. This allows adding new probes and also inserting probes into potentially compromised components *after* the suspected compromise, thus ensuring that the probes themselves have not been corrupted. At the same time, it is clearly impossible to always be collecting all possibly useful information, so it must be possible to activate/deactivate monitoring based on needs and resources. In essence, we need the ability to provide generic, low impact health monitoring, with the ability to turn up the intensity of monitoring when problems are suspected, and then once we're sure something is not a problem, we want to be able to turn that monitoring off and move on to something else.

Approach: OBJS' *Software Surveyor* is an independent software application that is capable of unobtrusively instrumenting and monitoring running distributed Java (and in the future C/C++) applications. The primary benefits of Software Surveyor include:

- The ability to compose views of the running distributed application describing the application's behavior, exceptions, and state. This information is used to render a visualization of the overall application, and as input to extensible validation mechanisms.
- Independent distributed dynamic monitoring - a more secure, less compromisable monitoring technique. Also alleviates the need for a significant persistent internal infrastructure for self-monitoring.
- Transient instrumentation - Does not increase the footprint of the on-disk application code. When the application terminates, only the original code remains.

In general, we want a family of generic monitoring tools that output in a common event schema and that can be accessed via a common event distribution system. This is a big engineering problem which is well on its way to being solved by DASADA. Use of generic monitoring means that the effort of building the probes and collection devices can be amortized. Dynamic insertion of probes (and the ability to turn them off) means that you can focus on the a suspected area or an area of interest without overburdening the system. If you collect everything available, you so burden the system that nothing gets accomplished. You don't want to embed probes or information-emitting code into each process because that would be unwieldy, wasteful and heavy-weight not to mention requiring much co-ordination in order to make sure that the information from different processes are compatible and able to be correlated. In terms of security, you'd like to be able to tap the line rather than bugging the phone itself. It makes it much harder for the intruder to spoof you or to disable your information collecting ability as well as harder to determine if he's been identified. Finally we need a family of analysis tools that will map configurations, measure resource utilization, correlate resources with program activity, track message flow, and detect historical patterns of behavior. We would expect to use OBJS' *Software Surveyor* along with some companion technology from the DARPA DASADA program. Already extant and in-the-works monitoring components of Cougaar can be made compatible by holding them to the common event schema and making use of the common event distribution infrastructure.

Technology Base: We begin from a rather substantial technology and design base developed under the DARPA DASADA program and tested in a TIE monitoring a loosely-coupled, internet-based intelligence gathering application. OBJS' *Software Surveyor* is a profiling toolkit to dynamically deduce and render the runtime configuration and behavior of evolving, component-based software. *Software Surveyor* v1.0, July 2001, (see screenshots) includes probes to collect a variety of information, an infrastructure to disseminate it, and synthesis tools to merge information streams and create dynamic configuration and usage maps. Information is synthesized from multiple sources and combined and rendered in a variety of formats and made easily accessible via the Web. Aggregated results identify "behavioral norms" to augment incomplete performance specifications and can be used with the probe infrastructure to signal users when the system is operating anomalously. *Software Surveyor* requires limited prior knowledge of application connectivity and can be used with dynamically reorganizing applications and those lacking complete specifications.



The *Software Surveyor* core is implemented in Java and can interact with probes and gauges in other languages, so *Software Surveyor* can be applied to Cougaar. We are currently 60% of the way through DASADA Phase I, so additional development will occur under this synergistic technology program. We are working on the following issues for the coming year's release. By the end of this year *Software Surveyor* will have been completely re-implemented improving upon ideas expressed in the first version. The principle features of this new version include a more advanced probe management infrastructure supporting both simple probes and more flexible probe stubs and plugs, a framework for probe configuration management, and framework support for both source code and bytecode probes. Finally, *Software Surveyor* now supports dynamic insertion and removal, and runtime control of probes in a running Java application (no other software that we're aware of is capable of doing this).

Issues: In concert with the core technology provided by *Software Surveyor*, the primary issues that need to be addressed include:

- A smooth integration/correlation of dynamically and statically collected information to form a coherent user-comprehensible picture.
- Context-sensitive "views" of the monitored system to provide information at a granularity of value to administrators/tools and consistent with their ability to take action. (Only provide information that can lead to an action)
- A robust notion of "focus" to allow selective monitoring and presentation.

A smooth integration/correlation of dynamically and statically collected information to form a coherent user-comprehensible picture is still needed. An architecture description language gives a template for an application. Once the application is actually instantiated and deployed, more details are filled in. Finally, during run-time, probes can provide details that were missing from the static information and details about the functioning of the actual application. The static model also helps determine probe placement.

Just as database architecture recognized the need for different views of the same data for different users, we need to have different views of the monitored events depending on our security rights and our life-cycle needs – depending on whether we're simply monitoring to identify intruders and take a pulse or whether we

are zeroing in on a specific problem by monitoring suspect nodes, programs, etc. Both the fish-eye view of the world from where you are and the helicopter view of the world that shows overall connectivity (the “big picture”) are required in differing circumstances.

Additional work needs to be done on focus. The first notion of focus is “aiming” – monitoring for different things at different times. We need the ability to collect different kinds of information in different scopes at different levels of granularity but need to limit the actual collection to those things actually helpful to the current need. We are currently implementing a probe management infrastructure to handle the mechanics of dynamic probe placement/activation/deactivation. The second notion of focus is “clarity of vision” – since we are not collecting everything, parts of the model/application are blurry (like a near-sighted person without glasses). The monitoring system needs to present the user with the blurred picture and the reasons for the blurs. It could be there is no information about a particular process because there was no information available, because the process isn’t actually running, or because we just weren’t looking at the information when it was available. Need to answer “What didn’t he know and why didn’t he know it?”

Objective: Provide independent monitoring and diagnostic mechanisms to the Cougar infrastructure to improve the understandability and reliability of Cougar-based applications. The mechanisms will:

- Identify potential/allowable interactions between application-level agents/nodes based on specifications, defaults, and initialization/deployment actions.
- Identify actual interactions between application-level agents/nodes based on runtime monitoring.
- Determine if the observed interactions are allowable based on specs, defaults, and initializations.
- Create local and system-wide maps of interaction patterns, and identify critical agents/nodes, choke points, and unused/underused agents/nodes.
- Provide visualization of monitored behavior and a comparison of this to expected behavior based on specifications.

These identifications, determinations and mappings provide needed inputs to the survivability mechanisms, policy decision-making mechanism, low-level policy implementers, and security mechanisms that will allow those mechanisms to make decisions. All of this must be consistent with Cougar architectural principles and security restrictions. The latter is essential, since otherwise the monitoring system (which, after all is expected to have a clear view of how the system is operating) becomes a massive vulnerability.

(E-6) White Paper – Ensuring the Robustness of Service Discovery Responses

OBJS White Paper to the DARPA UltraLog Program, 2002, Mark Greaves Program Manager.

Abstract: Description of how a variety of DASADA tools could be used within the DARPA UltraLog Program to ensure that dynamic binding of agents in consumer-producer relationships creates legitimate and survivable configurations with respect to declarative rules applicable to collections rather than simply individual bindings.

Ensuring the Robustness of Service Discovery Responses

Executive Summary

Cougaar Service Discovery based on the Cougaar MatchMaker is aimed at providing improved flexibility and robustness by allowing agents to *discover* service providers based on capabilities expressed in advertisements and service requests (queries). The set of agents identified in response to such a “binding query” can be used more or less interchangeably to satisfy the request, allowing load balancing and adaptation in the event of agent or infrastructure failure. While the individual agents identified as providing the requested service may be correct, adaptation requires that the set of agents be sufficiently large and robust that there are enough adequate alternatives and that the set of alternatives does not degrade subsequent to the initial service discovery attempt, leaving a requesting agent “high and dry” when it believes that it has fallback positions.

We believe the current plan for Cougaar Service Discovery should be augmented in several ways to become more robust. First, functional queries should be combined (via *query modification*) with extra-functional constraints (QoS and policy constraints) to ensure that queries result in the best result sets, taking into account all relevant factors. Second, we believe *query relaxation* techniques will be needed to locate acceptable matches when initial queries fail. Third, we believe an efficient scheme is needed to re-validate and rebind if services do not meet expectations or become unavailable. Fourth, to ensure that rebinding remains feasible, we believe that the set of potential options should be periodically revalidated.

We propose a collection of tools (compatible with the Cougaar architecture and time phased to match the projected Cougaar development schedule) to: (1) use policy statements to detect binding set inadequacies, (2) identify potential causes of such inadequacies, (3) suggest fixes, and (4) interact with Cougaar mechanisms to implement selected remedies.

Background

U*L CONOPS <https://docs.ultralog.net/dscgi/ds.py/Get/File-3283/CONOPS-v0.54.doc> requires a high degree of flexibility in creating and maintaining the command and support relationships that form the basis for all Cougaar agent interactions. The Service Discovery MatchMaker (MM) currently under development will ultimately make it possible to create these relationships by “query”, matching requests for services with advertisements of services that can be provided by the available agents. This will allow policies in the form of pre-packaged business rules to act as query modifiers (e.g., generalized constraints) to situation-specific specifications (e.g., unit location, OpTempo, command structure). The twin goals are: (1) quick setup for military missions with the ability to adapt as the military mission objectives change, and (2) greatly improved robustness of the overall system.

Speed and responsiveness are attained because relatively little situation-specific customization will be required to create or change a logistics planning configuration. Configurations can then be automatically converted into specific inter-agent connections. Improved robustness is achieved because capability-based connections and discovery provide logistics and planning alternatives so that neither: (1) the failure of the Cougaar infrastructure (e.g., agent or communications failure), nor (2) loss of logistics assets (e.g., supply depot blown up), nor (3) change of the logistics configuration (e.g., a unit moves away from its supply base and must get supplies from elsewhere) are catastrophic.

The Problem

All of the above benefits rely on the assumption that a query results in a suitable collection of service providers (agents). But what if that is not the case? How could it happen that the collection of service providers is not ‘suitable’? What could be done if it did?

To begin with, we assume that the following things work properly (this is not to say that other checking to ensure these properties is not necessary, just that those tools are outside our effort):

- MatchMaker correctly evaluates ontology-based queries using weighting functions specified in the query.
- The ontologies used by MatchMaker are correct.
- Agents that are found by MatchMaker either functionally do what they claim or the requesting agent can determine functional non-compliance and take corrective action.

In other words, we are assuming that any agent in MatchMaker’s response set will either be correct or that the requesting agent can detect a problem and ignore a functionally incorrect agent. Our concern instead is what happens if there is something undesirable about the *set* returned rather than with any individual member of that set. The sets could be poor in several different ways:

- The set is empty or contains only agents with too low a score to be acceptable. In this case, the binding attempt will fail. The problem could be caused by a logistics problem (e.g., there is no supplier of truck batteries) or an infrastructure failure (e.g., the agent for the part supplier is unreachable).
- Failure to bind may propagate back through a chain of agents, since if an agent cannot find a service, it may in its turn no longer be able to provide the services it is functionally capable of and thus would now be an unacceptable choice for some other agent that had planned to use it. For example, a wholesaling company may be able to sell both liquids and bulk solids. A military depot requests flour from it. The wholesaler tries to find a supplier for flour, but is unable to do so, so it is rejected as a source of flour, despite the fact that it is capable of delivering flour (if it could find it). The binding failure ripples back from the wholesaler-to-supplier connection to the depot-to-wholesaler connection. Further, the binding failure was caused by a parameter (flour) that flowed from depot to wholesaler to suppliers. Tracking this down on-the-fly will be tricky without automated support.

- Some members of the set are functionally correct, but have some other aspect that makes them undesirable. For example, a water provider may be too far away to be useful, but an incomplete specification of *mode aspects* fails to eliminate it. Worse, what happens if it becomes the top-ranked choice for some other reasons (e.g., advertised price), and gets used to the exclusion of a better alternative?
- The set contains too few alternatives to be robust. This causes two kinds of problems. First, just because an agent satisfies a binding request does not mean that it will be able to provide the service required of it in a given request during planning. For example, a warehouse might normally contain truck axles, so binding to it would be reasonable. However, when a particular request to ship truck axles is made, the warehouse may be empty. In this case, the requesting agent would like to have some potential alternate sources. The quality and number of such alternates should be specifiable by a higher level policy.
- Second, the set size shrinks *after* the query has been evaluated, leaving too few alternates. This could happen because of the loss of some of the alternates (e.g., the alternate warehouses get destroyed, leaving only the primary) or because changes in some mode aspect of the requestor or alternate providers (e.g., one or the other moves) makes a previously reasonable relationship become undesirable.

Undetected, the problems outlined above would make an application built on Cougar less able to meet the Ultra*Log robustness goals. Uncorrected by automated tools, the problems would increase the work-factor for administrators and slow down the configuration and repair of applications, making it impossible to meet the more aggressive Ultra*Log CONOPS adaptivity and ease of use goals.

The Root Causes

If MatchMaker works properly on correct ontologies, what can cause these problems?

There are (at least) four underlying sources of these problems: incorrect or contradictory business rules, finding the correct tradeoff between "precision" and "recall" in query specifications, insufficient logistics or Cougar resources, and propagation of constraints.

- Business rules will be used to express policy and constraints in many domains (e.g., purchasing regulations, security, product quality). This will require a large number of rules developed by experts in many domains. Getting this all correct and consistent is a daunting task. One has only to consider that the FAR and DFARS (which have been in existence for a long time and cover only contractual issues) are not internally consistent. Even if the rules are consistent, it is possible that they are too restrictive for a given situation: this is especially likely in degraded or emergency situations that the rules do not anticipate. In such cases, a rigorous application of all the overlapping rules may preclude doing anything at all. Humans can generally find a way around such problems; Cougar needs a similar ability to deal with the possibility of conflicting rules. This would be facilitated if it were possible to know which rules had been applied and which rules had caused alternatives to be rejected.
- Cougar uses weighted matching to support "approximate" or "ranked" matches. This is similar to a problem in the field of "information retrieval", in which queries

are used to identify possibly relevant documents from a (library) collection. A long-standing problem in that domain is the tradeoff between *precision* and *recall*; the ability to get *only* relevant documents and to get *all* the relevant documents. Cougaar faces the same issue; it is desirable to find all the relevant providers without introducing irrelevant providers. It is generally accepted in information retrieval that it is impossible to have perfect precision and recall simultaneously. Finding the right tradeoff between these two objectives, especially in a dynamic Cougaar environment where the set of potential providers changes rapidly, will be difficult. Thus, a mechanism to detect unsatisfactory result sets and tighten or loosen a query is necessary. This would be facilitated if it were possible to know, going in, what factors had contributed to the low quality of the result set.

- There may simply not be any (or enough) logistics resources providing the desired service with the required mode aspects. For example, a logistics robustness policy might state that there be at least three fuel providers in a theater, but what if there are only two? It is desirable that this deficit be detected during the matchmaking process so that either a waiver of the policy can be consciously made or an additional fuel source deployed.
- Matchmaking queries will be parameterized based on setup information (e.g., the physical location of the logistics unit represented by the agent) or on information passed transitively from another service request (e.g., a request by B to A to deliver truck axles to B causes A to look for a collection of C's that sell truck axles). Such (possibly transitive) parameterization is not subject to design-time verification and is a potential source of any of the above problems. It is a leap of faith to assume that all this will always work correctly

Nature of the proposed solution.

We propose a set of tools to specify policies related to binding set quality, to determine violations of those policies, to determine the cause of the violations, and to propose (re)solutions. All of these tools will be integrated into the Cougaar framework; in particular, they will interact with Cougaar Service Discovery, Adaptivity Engine, and Policy Management.

Specifically, we propose the following, integrated into Cougaar:

- Tools for defining and managing policy “meta-rules” describing desired and required properties of binding sets.
- A representative set of parameterizable meta-rules for common logistics policies as defined in Ultra*Log CONOPS.
- A tool to apply the policy meta-rules to MatchMaker output to determine if binding sets violate policies.
- Auditing tools to periodically determine if binding sets have changed to violate policy. Use of these tools may themselves be policy-driven (e.g., frequency of re-validation may depend on such things as criticality of the connection and likelihood of change to the set membership).

- A tool to present the various bindings of agents to system administrators so that these administrators can easily validate the reasonableness of the bindings and interactively take actions to react to problems.
- A tool to identify potential causes of binding set policy violations and suggest solutions in the form of modifications to queries, relaxation of rules, or deployment of additional logistics or Cougaar resources.

Details, Architectural Fit & Technology to Build On

The proposed tools are not intended to be either a replacement for, or a significant modification of, the planned Cougaar Service Discovery mechanisms. Instead, they are intended to pick up where Service Discovery leaves off. Note that instead of attempting to determine if the *individual* services discovered are adequate (the job of Service Discovery), we propose to determine if the *collection* of services discovered is adequate. This is an entirely different problem, but one that is essential to achieve robustness.

What follows is a list of the individual tools we plan to provide, their fit with the rest of the Cougaar system, and the technology starting point for each.

Tools for Defining & Managing Meta-Rules: We believe that the meta-rules can be represented in DAML-S as are other rules, advertisements, and queries in Cougaar Service Discovery. This will allow us to use existing and planned Cougaar tools for defining and manipulating the meta-rules. Very little effort will be required here.

Representative Policy Meta-Rules: Ultra*Log CONOPS describes scenario vignettes that can be used to define a set of such rules for use in testing. We will define rules applicable to the 2003 test cases.

Applying Meta-Rules to Binding Sets: Rules are predicates on properties of the set. We expect to be able to use an existing rule engine to apply these rules to the set returned by Cougaar Service Discovery. In the event this set does not satisfy the rules, an exception will be raised. This could lead to the application of an automatic repair strategy via the Adaptivity Engine, administrator intervention to take a corrective action, or simply a warning to the CSMART console.

Audit Tools: Because of the danger that an acceptable binding set may degrade over time, it is desirable to periodically re-evaluate such sets. The rule-application tool described above will be packaged (most likely as either a Cougaar Management Agent or Rover Agent) to do this. Because service discovery is potentially expensive, policy will be specified to determine the conditions under which re-evaluation takes place. Our belief at present is that this policy should take into account the criticality of the binding being supported (e.g., frequently used connections or connections responsible for highly critical parts of the logistics plan), the likelihood that the binding set will degrade below acceptable levels (e.g., a forward depot is more likely to be lost than a CONUS base and binding sets with many alternatives are more robust than those barely exceeding the threshold), the load on the Discovery Service (e.g., if the DS is busy making new bindings, it should not be further loaded with checking), and general infrastructure load (e.g., communications availability).

Administrator Interface: The checker (whether operating at discovery time or as part of background evaluation) fills the role of a Cougar sensor. Bindings associated with a given agent, node, or community, should be viewable graphically and textually. An administrator (possibly local to the agent, node, or community) will often be able to detect anomalies that slip past rule checkers. As such, the administrator should be able to signal exceptions through these tools in exactly the same way as the checking tool itself.

Identifying Causes of Binding Set Problems: This has several levels of sophistication, each requiring more effort. We anticipate providing these capabilities incrementally in roughly the order described here. At the first level, the checker will simply identify which rule or rules were violated. This requires no additional interaction with the Service Discovery MatchMaker and can be provided immediately. The next step requires extensions to the MatchMaker, so it will have to be done in conjunction with its developers. There appear to be three reasons that a binding set is inadequate: failure to satisfy the binding query, weightings and thresholds that reject otherwise acceptable response items, and failure to satisfy the business rules that condition the query evaluation. The ways in which these are identified appear to be different.

- Failing to satisfy the binding query. In this case, the collection of agents whose advertisements satisfy the query predicate is insufficient. If the query predicate is complex, it may be difficult to determine why this occurs. Related work done in the mid-1980's to mid-1990's in the database field developed techniques to figure out what part of a query caused anomalous results by tracking the sizes of intermediate result sets computed during the evaluation process [work on *query modification* in Ingres and System R, work on *cooperative response* by several groups, work by Jonathan King and Gio Wiederhold at Stanford on *semantic query optimization*, and work on *query relaxation* by Wesley Chu at UCLA]. Points in the evaluation where set size either exploded or shrank to an unexpectedly small size (or in the Cougar case to below the acceptance threshold) were flagged, with the part of the query being evaluated at that point being a suspected source of the problem. The system was not foolproof, but did help with the identification process. This might be able to be adapted for Cougar, the principle changes being that Cougar queries and advertisements are not relations and that query weighting needs to be considered. This would require MatchMaker to expose the evaluation order and statistics on the intermediate results. It would not require any MatchMaker modifications beyond this increased visibility.
- Weighting and threshold problems. This may be partially handled by the planned Service Discovery weighting scheme, but the idea is to identify any weighting terms that are consistently not satisfied (e.g., the highly weighted term "US flagged carrier" is never satisfied, causing no carriers to be found with sufficiently high weight to pass the threshold). These should be suspected as flawed or overly restrictive. Again, MatchMaker visibility is required to determine this information.
- Restrictive business rules. Agents that satisfy the query predicate may be rejected due to combinations of business rules. If MatchMaker exposed statistics on the number of potential responses that violated any business rules that fired, it would be possible to identify potential business rule conflicts or rules whose application might

be too restrictive for the situation. These could either be handled by an override policy or thrown to an administrator via CSMART.

Suggesting Solutions: The information that can be collected above may imply a solution to a human administrator,. However, the data sets involved may be of such a size as to make determining such a solution infeasible. We propose to build tools that can make this easier, and may even allow certain first-level solutions to be automated. Grouping and ranking the potential sources of binding set inadequacy according to impact and likelihood would help make sense out of a potentially large information collection. What-iffing tools could allow checking the effect of a proposed changes to query, rules, policies, or weighting/thresholds on the binding query result without causing an actual bind to occur. This would make use of the Service Discovery without changes, but the originator of the request would be the what-iffing tool rather than an agent trying to bind. An issue would be managing the load that this creates on Service Discovery, since massive what-iffing could prevent actual bindings from taking place. If Service Discovery optimizes queries, the query evaluation plan may not have an obvious correspondence with the original query (since the optimal evaluation order may differ from the order in which the predicate was expressed). If this is the case, it will be necessary to either recreate this correspondence or interact with MatchMaker to cause it to be exposed. This is a more difficult problem, but the DBMS query modification work mentioned above resolved this issue, at least in the relational database context.

Implementing Solutions. Once a corrective action is decided upon, whether by a human administrator or by an automated policy, actions need to be taken. This requires making some change to the conditions that caused the failure and then re-executing the discovery attempt. An issue in the re-execution is whether the requesting agent must be aware of this or whether it can happen transparently. Correcting the causes of the failure naturally requires connections to other parts of Cougaar. Actions that could be taken include:

- Change the query predicate, weightings, or threshold. This would require interaction with the requesting agent. As such, an additional interface for agents would likely need to be defined. An issue is whether the change would apply to only the current discovery attempt or to all future discovery attempts.
- Change the current policy or override a portion of the policy. An issue is the scope of the change (does it apply only to the current request, the requesting agent, all agents of that type, all agents in that community, globally?). This kind of change should be able to be made by an administrator.
- Deploy (or change the properties of) one or more logistics resources. This requires reaching outside of Ultra*Log.
- Change (fix) a business rule. This is a heavyweight action and should not be expected to be made on-line.

Competing approaches

Cougaar does not currently provide capabilities such as those described above. The Ultra*Log Service Discovery group is not currently discussing these issues. Cougaar

Policy Management might specify policies to respond to deficiencies in binding sets, but would still require mechanisms to detect and categorize such problems prior to policy application. The Cougaar Adaptivity Engine likewise represents a way in which to respond to problems but does not itself detect or categorize the problems.

Deliverables Schedule

2003

In 2003, we will be developing tools that allow detection of problems with the original binding sets created by Cougaar Service Discovery. Determining the causes of problems will be deferred until 2004. This is for three reasons:

- This allows us to get the basic mechanisms and architectural fit correct before attempting to be too sophisticated.
- Work proposed for 2003 requires only the ability to examine the standard outputs of the Cougaar Service Discovery MatchMaker, whereas MatchMaker extensions to expose some aspects of its internal operations are required for the advanced capabilities. The Discovery Service is still early in its development and is operating on a different development cycle than most of the rest of Ultra*Log (having started in summer). We believe that we are more likely to get significant design cycles with the Service Discovery developers once their basic capabilities are completed.
- A smaller initial effort allows us to demonstrate an important capability at relatively low cost with a roadmap toward a more comprehensive capability in 2004 and beyond.

Specific capabilities to be delivered for integration into the 2003 Cougaar Release are:

- Tools for defining and managing policy “meta-rules” describing desired and required properties of binding sets.
- Representative policies for the properties of binding sets.
- A tool to apply the policy meta-rules to MatchMaker output to determine if binding sets violate policies.
- A tool to present the various bindings of agents to system administrators so that these administrators can easily validate the reasonableness of the bindings and interactively take actions to react to problems.
- Connection to either the Cougaar Discovery Service or Adaptivity Engine (whichever is most appropriate) to take corrective actions in the event a binding set is inadequate. Most responses in the first year will either be simple or will involve administrator actions.

2004 and Beyond

In 2004 and beyond, we will provide:

- Audit tools to periodically determine if binding sets have changed to violate policy.
- Representative policies to drive the audit tools.

- A tool to identify potential causes of binding set policy violations and suggest solutions in the form of modifications to queries, relaxations of rule, or deployment of additional logistics or Cougar resources.

The specific deliverables and schedule after the first year depend on Ultra*Log needs as discovered over the course of work.

(E-7) Report on Demonstrations of ProbeMeister Technology to the UltraLog Program

Excerpt from the February, 2002 OBJS Monthly Progress Report from the DARPA UltraLog Program.

Abstract: Describes demonstrations of three kinds of uses of ProbeMeister within the UltraLog program for Red Teaming, distributed stress injection and data collection, and distributed debugging of a messaging subsystem.

Excerpt from February 2003 MsgLog Progress Report To the DARPA UltraLog Program

Progress. In advance of the CDR and our planned meeting there with Mark Greaves, we prepared three demonstrations of ways in which OBJS' ProbeMeister (PM is a dynamic software instrumentation tool for Java developed in the DASADA program) could be of use on the UltraLog program. One demo showed how PM could be used as a Red Team attack tool, allowing infiltrators to modify agent blackboard contents unbeknownst and thereby modify the result of planning. Another showed how PM could be used by the Engineering Test and Assessment teams to inject stresses and monitor defensive actions without requiring any cooperation from the developers (no Cougar Events). The third demonstrated our MsgAudit tool as an example of how to use PM for regression testing and debugging, even allowing dynamic code insertion and monitoring of the Java, thereby facilitating debugging control flows that cross the application/JVM boundary and those that involve multiple JVM's and hosts, as Cougar does. We showed these demos to several people at the CDR, including Mark Greaves, Eric Rickard and Manoj Srivastava, and expect at least SRI, STDC, and Sandia to evaluate PM for use in their 2003 work.
