Web Services Navigator: Visualizing the execution of Web Services

W. De Pauw
M. Lei
E. Pring
L. Villard
M. Arnold
J. F. Morar

The Web Services standard is becoming the lingua franca for loosely coupled distributed applications. As the number of nodes and the complexity of these applications grow over the coming years, it will become more challenging for developers to understand, debug, and optimize them. In this paper, we describe Web Services Navigator, a visualization tool that fosters better understanding of service-oriented architecture (SOA) applications. We draw on our experience with real SOA applications to show how this tool has been applied to practical problems ranging from business logic misunderstandings to performance bottlenecks to syntax and semantic errors. Web Services Navigator helps to solve these problems by visualizing how applications really execute, enabling business owners, application designers, project managers, programmers, and operations staff to understand how their applications actually behave. We sketch the architecture of Web Services Navigator, outline how it reconstructs application execution from event logs, and describe how users interactively explore their applications using its five linked views.

INTRODUCTION

Significant portions of the productivity gains enjoyed by businesses over the past decades are attributable to the adoption of new information technology (IT). At some point the economic balance shifts; businesses start putting more emphasis on reducing the cost of supporting existing IT functions than on adding new function. Today, many businesses are striving to improve the overall cost-effectiveness of their IT investments by reviewing business needs and cutting costs. These efforts typically include leveraging existing assets, consolidating redundancies, and laying a foundation for future growth. This trend is fueling the move from tightly coupled componentized systems to loosely coupled service-based systems, such as those based on service-oriented architectures (SOAs) employing standards-based interfaces.[1,2]

To illustrate the differences between componentized systems and service-based systems, we make an analogy with the air transportation industry. This industry moves passengers arriving by means of ground transportation into airplanes, flies them to a

destination, and discharges them into ground transportation at the destination. Airplanes, an essential part of the story, are constructed from large collections of tightly integrated components. According to Boeing, a single 747-400** airplane contains more than 6 million parts,[3] demonstrating that the componentized system approach can work for very complex systems. Even so, the overall air transportation system is best conceptualized as a composition of "choreographed" *services*: fuel services, food services, baggage services, air traffic control services, and ground transportation services, to name just a few.

Air transportation depends on both component strategies and service strategies. What forces drive businesses toward one approach or the other? One way to address this question is to observe some of the differences between integrated components and choreographed services.

Componentized systems tend to be deeply hierarchical, with components constructed from subcomponents that are themselves constructed from other subcomponents. When a componentized system malfunctions, the misbehaving subcomponent must be identified and repaired or replaced. For example, frequent airline passengers may be familiar with delays associated with airplane malfunctions. Drilling down into such "vertical" component hierarchies to identify the root cause of failure is the strength of today's diagnostic tools, which are often built into complex systems.

How does this contrast with the choreographed services environment? Services have well-defined and broadly available interfaces with a predictable or specified quality of service. They are typically accessed by different types of consumers. Services generally provide a complete usable unit of work, and equivalent services are frequently available as alternatives. Indeed, interchangeable services are the basis for competition. When a service fails, the preferred remedy may be substitution rather than repair. For example, if a fuel truck breaks down, any airline passenger would expect a replacement truck to be dispatched rather than waiting for the broken truck to be fixed. Unlike componentized systems, the domain of control and visibility for the consumer of the service generally stops at the service interface. This service environment leads to numerous "horizontal" interaction patterns that take the form of a broad and shallow hierarchy.

The challenge in managing and understanding service composition is dealing with this horizontal complexity. When systems are composed of multiple independent business processes, the mapping between such processes and the applications executing in the IT layers may not always be obvious. Consider such fundamental tasks as verifying the correctness of a workflow or locating performance bottlenecks, where logs from a variety of servers must be collated and interpreted. This is difficult even when the number of servers is small and may become impractical as horizontal complexity increases.

The emergence of SOAs in the IT industry is driven by the same pressures that have shaped the air transportation system. Web Services Navigator[4] has been designed and built from the ground up to address the challenges of horizontal complexity in the service environment and its associated patterns of usage. This tool relies on the Data Collector for IBM Web Services Navigator[5] to capture events. The tool then correlates the events, models the transactions they represent, and extracts patterns of execution from the model. The tool thus reconstructs individual transactions from end to end and produces visual abstractions, such as service topologies and flow patterns, as well as concrete views of transaction flows and data content. The abstract views reduce large volumes of execution data to forms more meaningful to humans. They can reveal important business trends that are obscured in concrete views and at the same time, can link to the associated detailed data when needed.

Because SOAs conform to open standards, the Data Collector for Web Services Navigator can capture application execution data in the middleware. Consequently, developers and operations staff can take advantage of visualization technology without modifying their applications, either before or after deployment, even when their environment includes a mixture of programming languages and operating platforms.

In the next section of this paper, we describe the five views that are used by Web Services Navigator to display information. The two sections that follow illustrate a number of cases in which the tool has

proven useful; we show first how the tool was used to diagnose a number of problems in business logic, and then we show a number of cases in which the tool helped diagnose problems in the IT layer. The problems presented in this paper were encountered in real Web Services-based applications developed within IBM or for customers of IBM. In particular, two applications were used to illustrate the tool features: an online order validation service and a retail point-of-sale system. Next, we briefly discuss the Data Collector. Then, in "Architecture and implementation," we describe the general architecture of the tool and discuss a number of implementation issues of interest: compensating for clock skews among nodes, our algorithm for laying out topology graphs, and identifying transaction patterns. We discuss related work, and we conclude with a summary.

The Web Services Navigator described in this paper is a research project. It is publicly available from the IBM alphaWorks* Web site and is included in a recently released product from IBM.[6] A previous version is also available from IBM.[7]

THE VISUALIZATION TOOL

The Web Services Navigator visualization tool is a plug-in feature for the Eclipse Version 3.0 platform,[8] an open framework for interactive tools. Web Services Navigator offers a new perspective with five new views of the execution of Web Services applications, as shown in ***Figure 1***. It can co-exist with, but does not depend upon, other tools[9,10] and products[11] based on the Eclipse platform.

This section describes the five new views presented by Web Services Navigator: Service Topology, Transaction Flows, Flow Patterns, Statistics Tables, and Message Content. Examples of how each view visualizes problems are illustrated in later sections of this paper and in a companion article[12] published on IBM alphaWorks.

The Service Topology view, on the upper right side of Figure 1, shows a graph of the services involved in an application and the message flows between them. The boxes represent individual Web services: each is labeled with the name of the service (at the top of the box), the name of the machine that hosts the service (below the service name), and the names of one or more operations that the service provides (shown in gray background). The colors of the boxes indicate different host machines, and all services hosted by the same machine are displayed in the same color. The arrows between boxes represent Web service requests and are labeled with the number of requests captured. The circumstances that produced this particular view will be discussed in a later section.

When the cursor (mouse pointer) is moved near a node or an edge, information such as the total number of requests and the total number of bytes transmitted is displayed in a small "tooltip" window overlaid on the view.

The Transaction Flows view, on the left side of Figure 1, shows a time sequence of the Web services involved in an application and the message flows between them. The columns represent individual Web services and are labeled with the names of the services and the machines that host them. Each service and host machine is labeled in the same color as the Service Topology view. Arrows represent Web service requests and responses, and vertical bars represent processing of a request. Transactions are separate flows of connected arrows and vertical bars. Each transaction is drawn in a unique color. The colors of the labels and transactions are not related. Time proceeds from top to bottom; it may be represented conventionally, in seconds, with a scale running down the right-hand side of the view; or, time may be represented in logical steps, in proper sequence but with all events evenly spaced, regardless of their actual duration. The logical representation of time may reveal details within dense clusters of events, and across widely spaced events, that are obscured in the conventional representation. Additional information can be displayed in tooltip windows by moving the cursor over service names, machine names, message arrows, and processing bars.

The Flow Patterns view, illustrated on the right side of ***Figure 2***, may provide more insight into the behavior of an application than the Transaction Flows view when it is cluttered by many concurrent transactions.

By automatically classifying similar transaction flows into flow patterns, Web Services Navigator significantly reduces the amount of information that

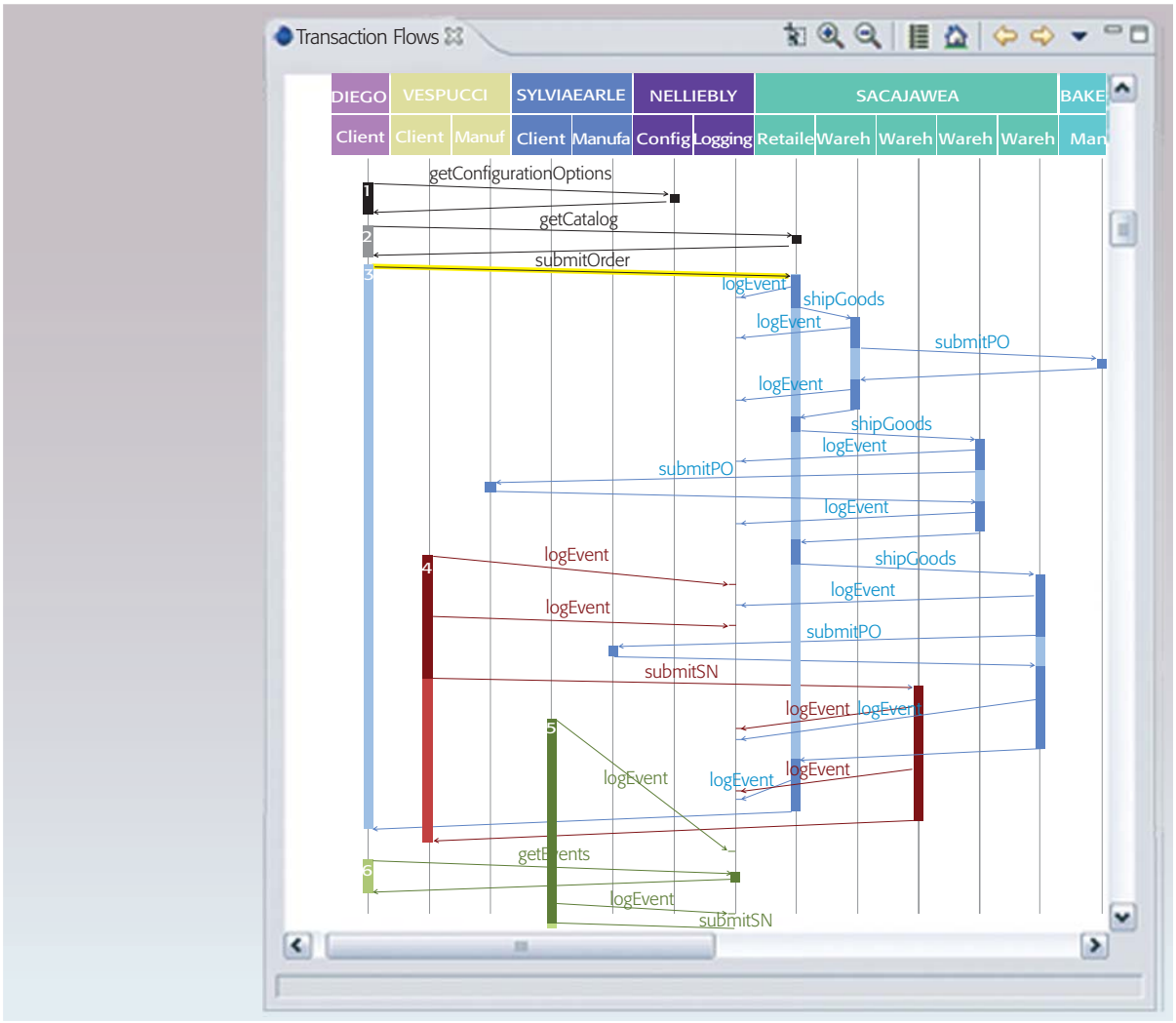


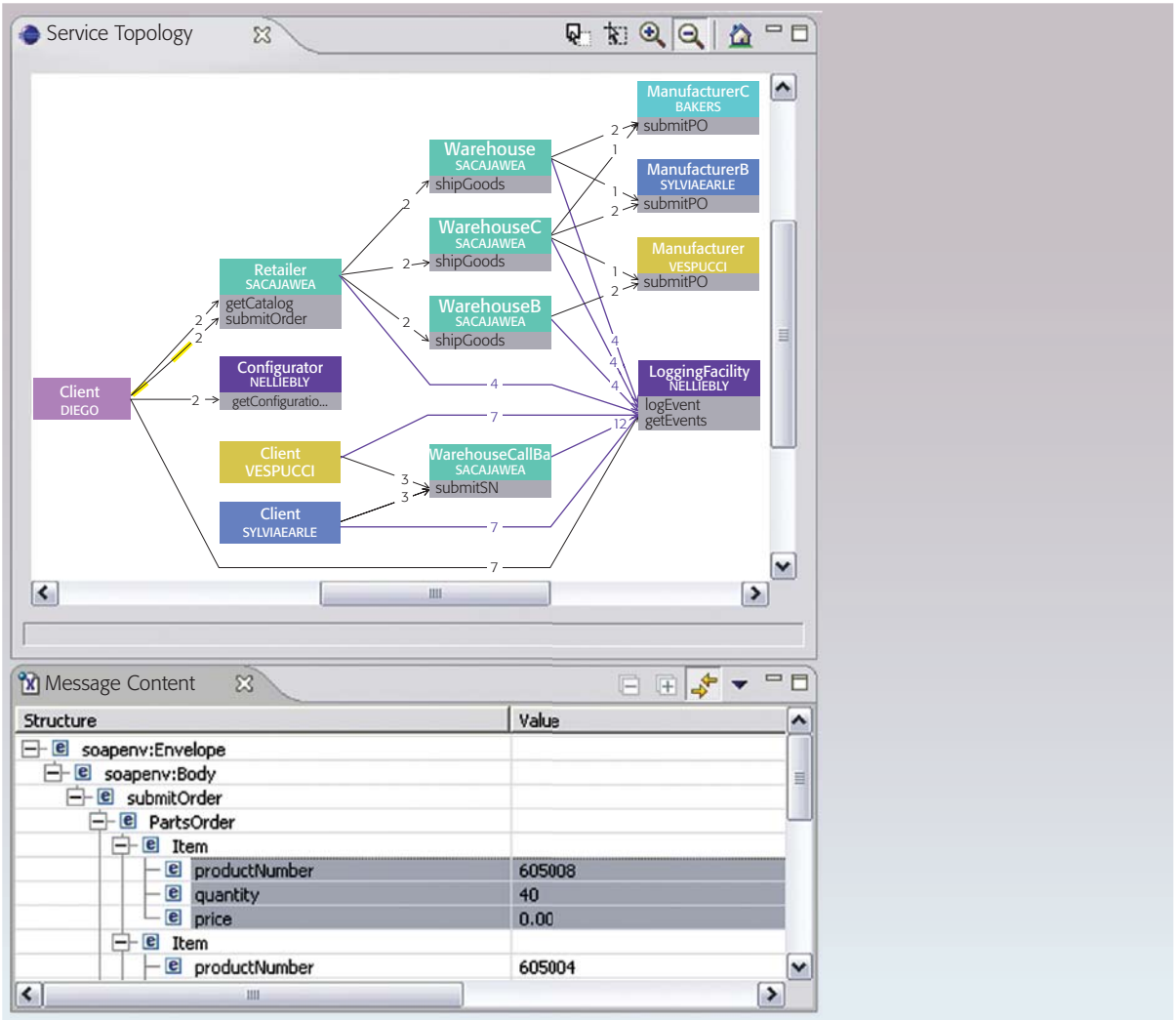
Figure 1
The Web Services Navigator shows multiple views of application execution

a user must digest. Patterns are based on similarities between the order of the invoked services, regardless of the exact timing. Patterns reveal common trends, as well as abnormalities or outliers, enabling users to explore their applications at different levels of abstraction. This is explained in more detail in the section "Architecture and implementation."

For example, the pattern labeled "Pattern 1: 70 x" in the Flow Patterns view represents 70 similar transactions in the Transaction Flows view.

The Statistics Tables view, illustrated in ***Figure 3***, shows execution performance information numerically with four levels of granularity:

- The *message statistics* table includes data on individual messages (request or response), including the name of the sender (service and machine), name of responder, network transit time, and message size.
- The *invocation statistics* table includes data on request/response pairs, client and server elapsed times, network delay, server processing time, and message sizes. For each invocation, the server elapsed time and the network delay add up to the client elapsed time. This table is shown in the lower part of Figure 3.
- The *transaction statistics* table includes data on trees of connected invocations, usually starting at a client. The data, which pertains to the initiating service and machine, include the number of



invocations and the number of services and machines involved.

- The *pattern invocation statistics* table includes data on groups of similar transactions, such as averages, minimums and maximums for elapsed client and server time, network delay, server processing time, and request and response message sizes.

Typically, users may start exploring the captured data through the first three views (Service Topology, Transaction Flows, and Flow Patterns) and after pinpointing a problem, may use the Statistics Tables view.

The Message Content view, on the lower right side of Figure 1, shows the contents of a selected message in an Extensible Markup Language (XML) viewer. When the user clicks on a message in one of the other views (such as the Transactions Flow view or the Statistics Tables view), the contents of that message are automatically displayed. The Message Content view can show the entire contents of the selected message, or, when Simple Object Access Protocol (SOAP) headers and XML namespace identifiers are not helpful, they can be hidden to simplify the view. Because Web Services Navigator is an Eclipse plug-in, it can display multiple views of the same data at the same time. When a message is selected in any view, that message is highlighted in

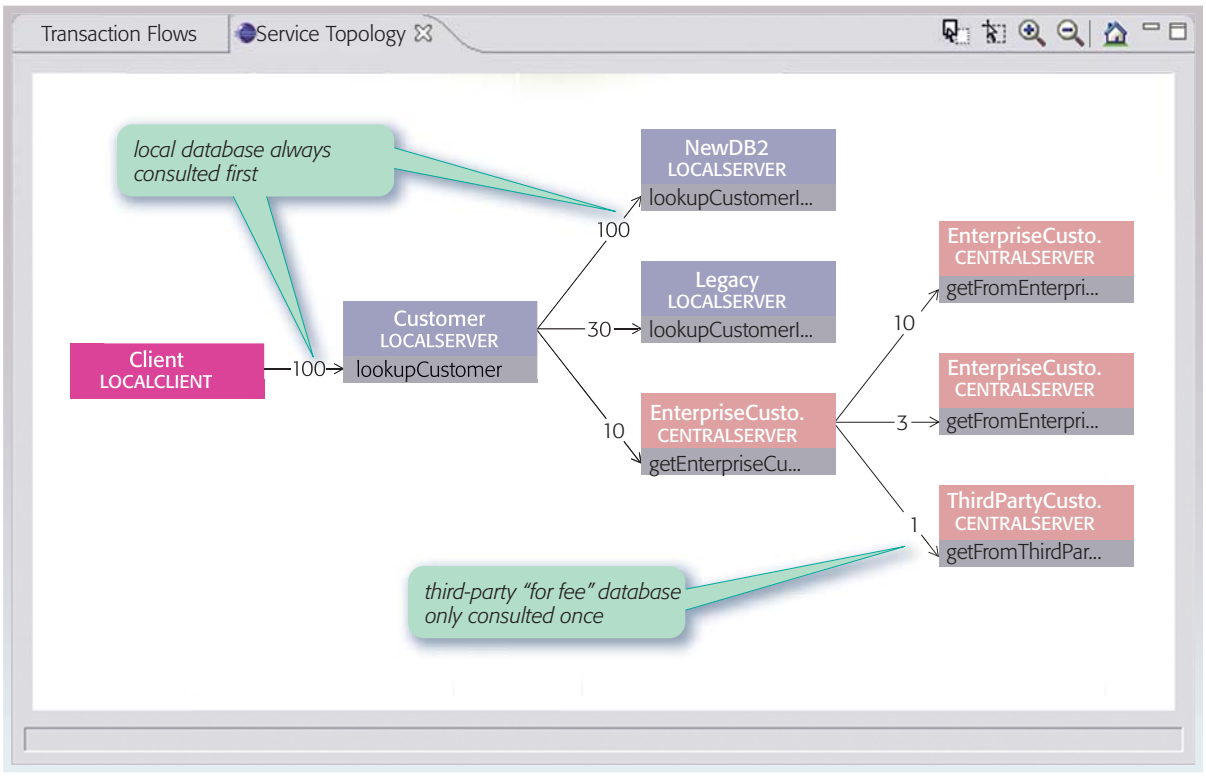


Figure 2
Verifying access patterns to a remote database for a retail point-of-sale system

yellow in all views, and its payload is automatically displayed in the Message Content view, as illustrated in Figure 1.

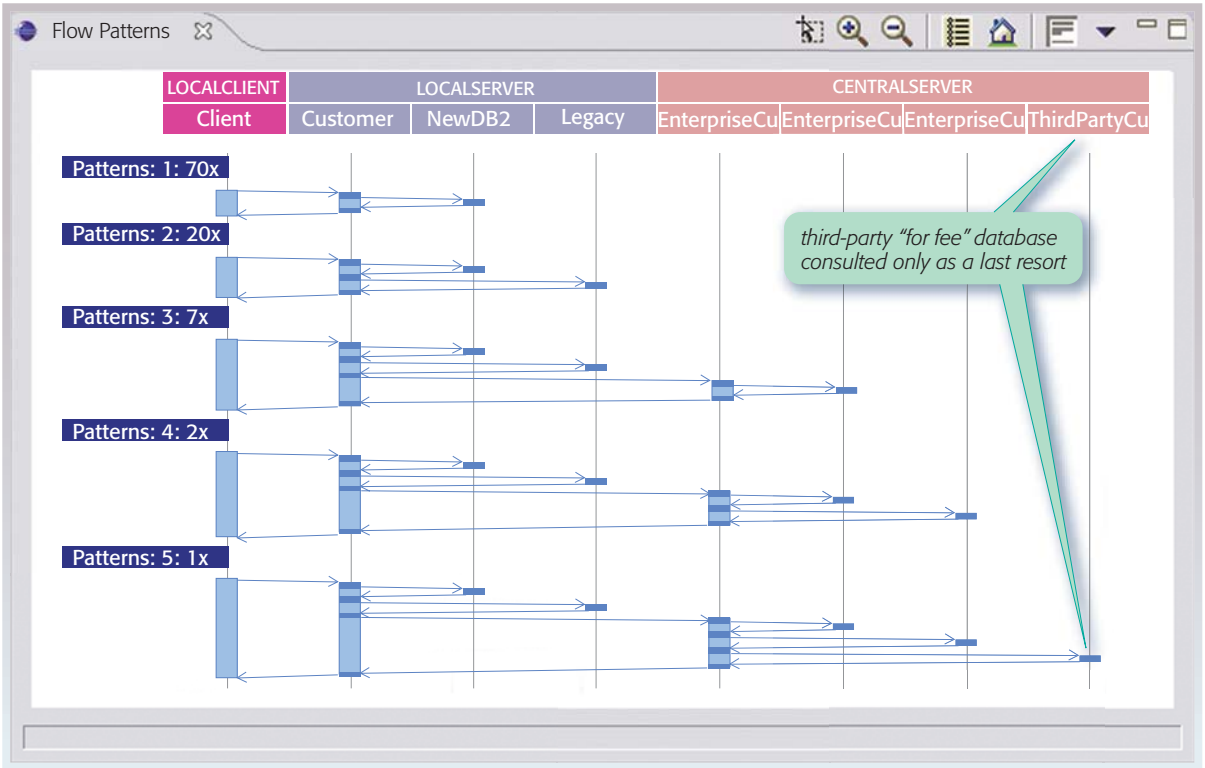
VISUALIZING PROBLEMS IN BUSINESS LOGIC

Web Services Navigator helps business owners, application developers (architects, programmers, project managers, and testers), and operations staff to understand the behavior of their Web Services-based applications and diagnose problems through visual representations of their execution. This section presents several practical problems in business logic that we have explored with the tool: verifying workflow choreography, detecting incorrect implementation of business rules and excessively "chatty" communications, and verifying application cost structure. The problems presented in this and the following section (which deals with problems in the IT layer) were encountered in real Web Services-based applications developed within IBM or for customers of IBM.

Verifying workflow choreography

The architecture of a business process may appear simple when it is being designed in a workflow modeling tool. In practice, however, when the architecture is implemented across organizational boundaries and deployed across geographical boundaries, verifying that the implementations faithfully realize that architecture is not simple at all.

The Supply Chain Management sample application published by the Web Services Interoperability (WS-I) organization is an example of such a workflow.[13] Its architecture is straightforward. A Retailer system includes catalog-search and order-submission services accessible to online customers, and warehouse shipping request and notification services available only internally. Manufacturer systems provide wholesale purchase and delivery services to the Retailer's warehouses.[14] This architecture has been implemented by at least 10 different organizations, and extensive testing has demonstrated that all of the implementations of each service can successfully interoperate. For example, IBM's implementation deploys a Retailer system with three warehouses and three Manufacturer systems on WebSphere* Application Servers.



Web Services Navigator presents multiple views of the actual execution of the deployed application, permitting each user to examine aspects of the application of interest to them, as illustrated in Figure 1. The Services Topology view (in the upper right side of the figure) allows business owners to confirm that the Retailer system is communicating with all of its warehouses, and that the warehouses are communicating with manufacturers. The Transaction Flows view (in the left side of the figure) allows architects to verify their logic by following the message flow of individual transactions. The Message Content view (in the lower right side of the figure) allows programmers to inspect the structure and content of individual messages. The views are linked together so that users can highlight messages of particular interest and drill down into details of particular concern.

Detecting incorrect implementation of business rules

The business owners who commission SOA applications often speak a very different language from the developers who implement their requirements. In spite of their best efforts to forge a common understanding, misunderstandings sometimes ensue between business owners, application designers, and programmers that result in the incorrect implementation of business rules.

One such misunderstanding occurred in the implementation of a service for order validation in an online application. The implementation of this service consisted of two distinct workflows—one for new orders, the other for follow-on orders. The initial implementation seemed to handle both order types correctly. However, the business owner, who was not involved in the technical aspects of the project and had no prior training with Web Services Navigator, immediately spotted a problem in the visualization of an apparently successful test, illustrated in *Figure 4*. The business owner recognized that for a new order, the invocation of the StackingChecker service (identified by the text balloon in the Service Topology view) was incorrect: the business rules specified that this service should be invoked only for follow-on orders.

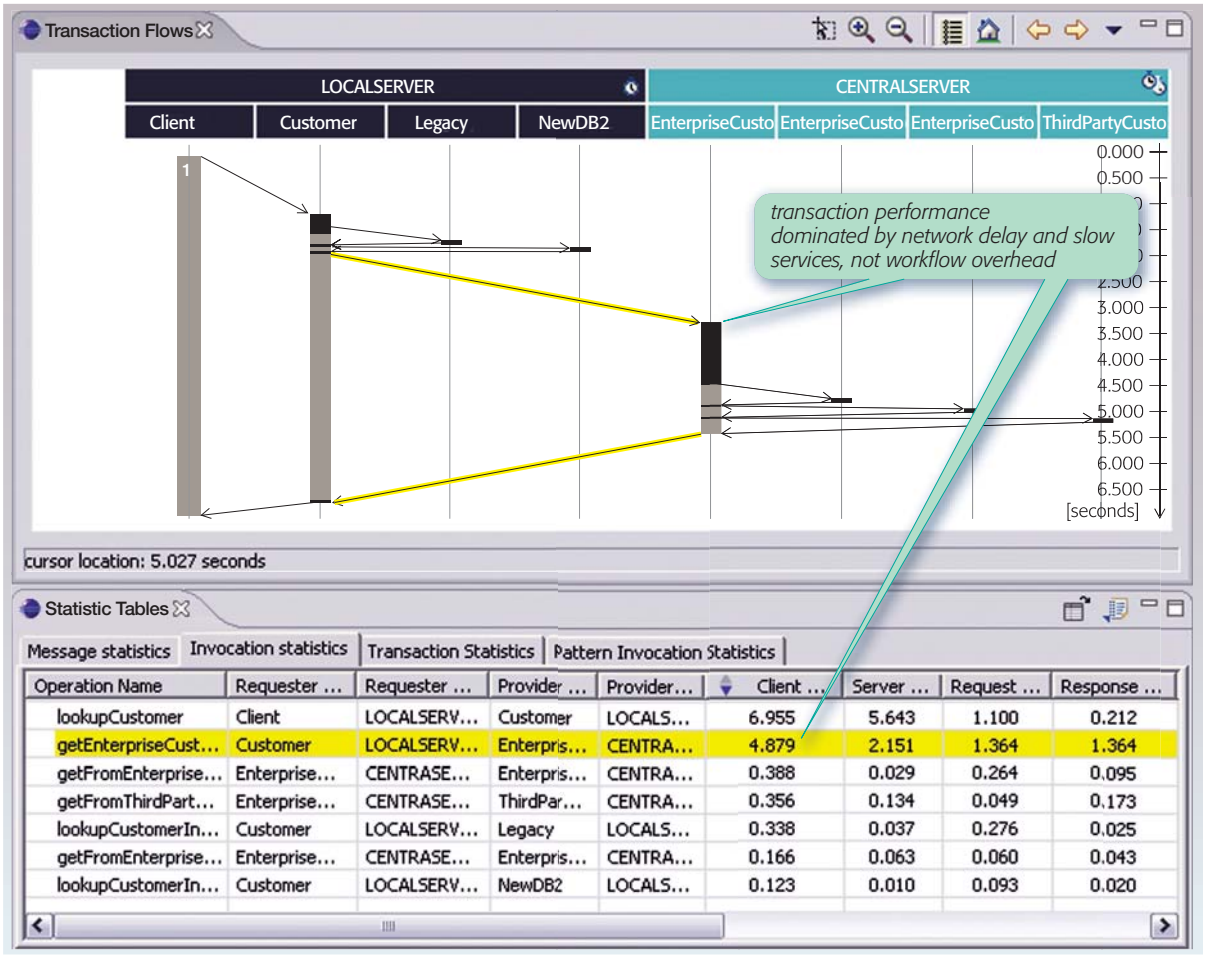


Figure 3
Identifying network delay and slow response from Web services as the cause of performance bottleneck

Detecting excessively chatty communications

As soon as the implementation of a distributed application reaches the early testing phase, Web Services Navigator can be used to verify its correctness and identify performance bottlenecks. One characteristic to look for is excessive or unnecessary communications, because service invocations tend to be performance intensive and transit time between different machines may be significant.

For example, the follow-on order transaction for the online order validation service mentioned earlier permits clients to submit requests containing multiple line items. Interactive clients, for which the initial implementation was modeled, typically submit requests with only a few line items. However, when new test data was supplied that included submissions from batch clients, requests with hundreds or thousands of line items appeared.

The initial implementation handled large transactions correctly but slowly. The "Find repetitions in invocations" function of the Transaction Flows view, illustrated in ***Figure 5*** by the green rectangles labeled "4X", showed that the application was iterating through order line items by invoking four times the same sequence of services separately for each line item. Each of the four repeated sequences is framed by a white rectangle inside the green rectangle indicating the total repetition. A subsequent implementation invoked the services once with a list of line items, rather than invoking the services repeatedly, thus dramatically reducing overhead for batch clients and improving response time for interactive clients.

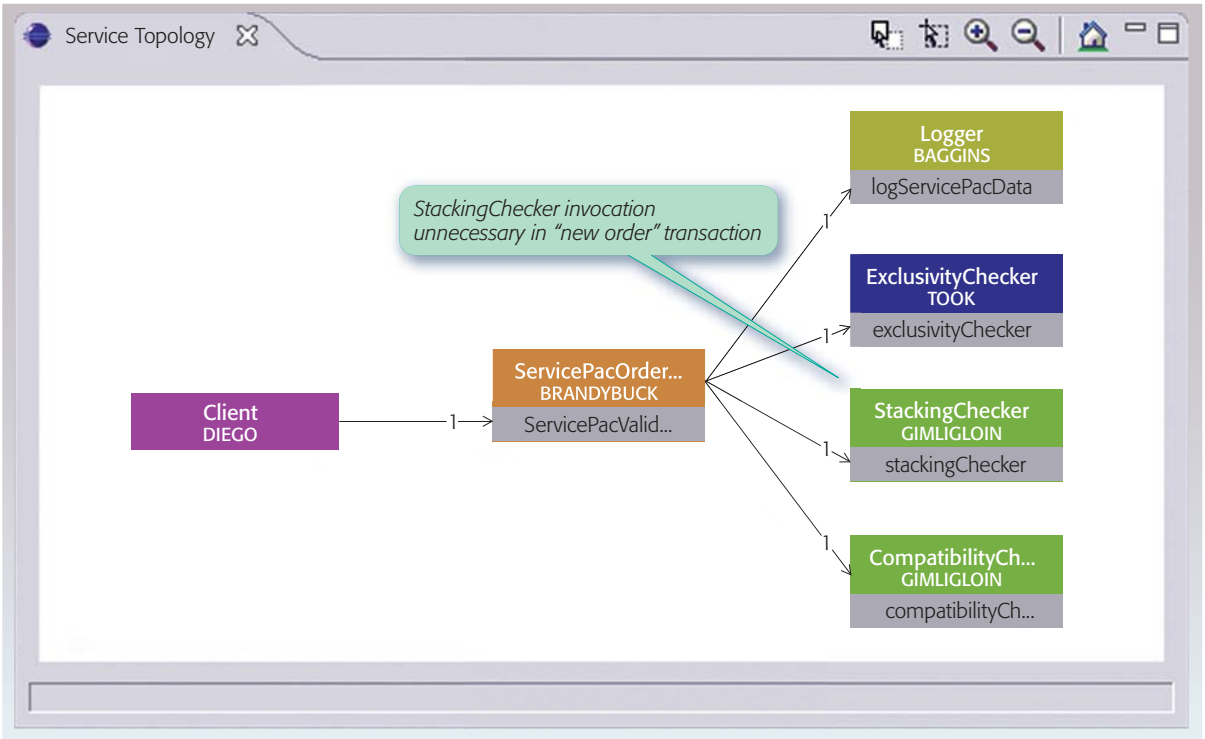


Figure 4
Detecting an unnecessary service invocation in an online order validation service

Verifying application cost structure

The business owners who commission SOA applications are often as concerned with minimizing the cost of operations as they are with ensuring the correctness of the logic. The costs of greatest concern may be bandwidth on their private network, processing workload on their servers, fees paid to other parties, or a combination of these factors.

One such SOA application is a customer information search for a retail point-of-sale system. The business owner wanted to minimize network bandwidth on the relatively slow links between retail stores and the corporate data center, while searching for information about known customers in a new local database engine before burdening a central mainframe. The application was permitted to purchase information about unknown customers from a third-party service, but the business owner wanted to ensure that this option was exercised only after all in-house options were exhausted.

Web Services Navigator enabled the project manager to verify that the application faithfully implemented this cost structure. In Figure 2, the Services Topology view showed that there was only one request to the "ThirdPartyCustomerDB" service in a sampling of 100 transactions (indicated by the request counts on the arrows, as identified by the left-hand text balloons in the figure). The Flow Patterns view showed that this request was made only after all other databases had been queried (indicated by the order of its service invocation, as identified by the right-hand text balloon in the figure).

For business owners, patterns can be used to verify the correctness of a business process, and they can show trends in business activities. For architects, patterns can show how services typically interact with each other, enabling them to compare actual behavior with the original design. For programmers, patterns can help with optimization and debugging by distinguishing the most frequently occurring patterns from outliers and by identifying bottlenecks that consume inordinate resources. For testers, patterns can identify intermittent abnormalities and outliers in the huge volumes of data produced by applications under heavy load.

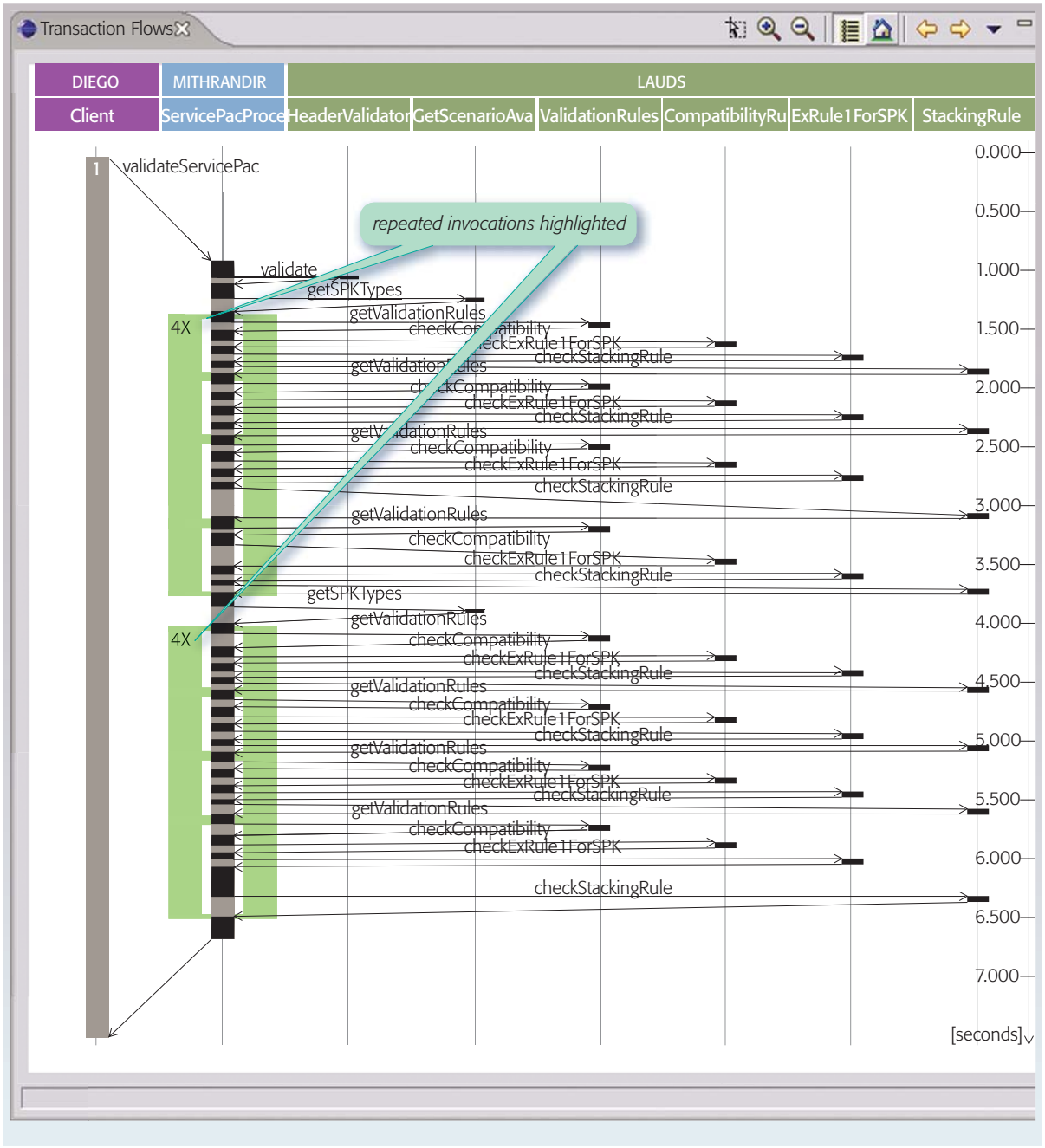


Figure 5
The use of Find Repetitions in Invocations function to detect excessively fine-grained communications in the online order validation service

VISUALIZING PROBLEMS IN IT LAYERS

In addition to problems in business logic, as described in the preceding section, Web Services Navigator can also visualize problems in the IT layers of Web Services-based applications. This section presents several problems in the IT layers of applications that we have explored with the tool: incorrect implementation of service semantics, transaction bottlenecks, unavailability of resources, and syntax errors in message encoding. The problems presented in this section and the preceding section were found in real SOA applications developed within IBM or for customers of IBM.

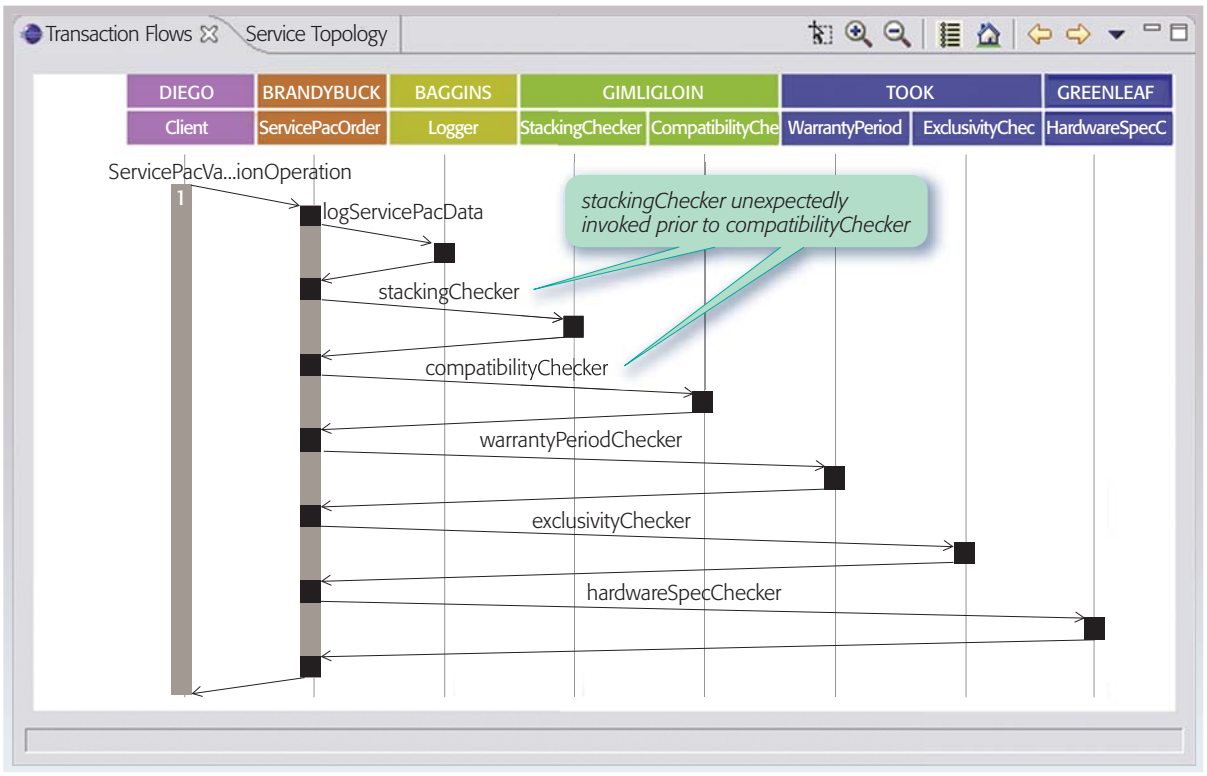


Figure 6
Detecting the incorrect invocation order of two service calls in the implementation of the order validation service

Semantic errors

The designers of SOA applications typically compose their applications with Web Services developed by other groups in different organizations by using middleware provided by systems vendors. Within this complexity lurk countless opportunities for misunderstanding the detailed semantics of the components that the application depends upon.

The business logic for the online order validation service mentioned earlier specified the steps to be executed for a follow-on order transaction, and the application designer coded these steps using a workflow language. During initial testing of the application, the designer noticed, in the Transaction Flows view illustrated in *Figure 6*, that the StackingChecker step was occasionally executed out of sequence. Functional testing did not catch this error because the workflow steps were implemented with stub services. The documentation for the workflow language specified that steps were by default considered to be concurrent unless explicitly marked to be serial. A review of the workflow engine revealed that it scheduled concurrent steps sequentially and usually, but not always, in the order they were coded. The problem was fixed by adding explicit coding for serial execution of the workflow steps.

One of the users of the online order validation service was a multithreaded client driven by a batch process. During initial testing of the client, the designer noticed, in the Transaction Flows view illustrated in *Figure 7*, that the client occasionally sent two nearly simultaneous requests, causing the workflow engine to launch two parallel transactions. The Message Content view confirmed that the two requests were identical (both request messages contained the same value in their serialNumber fields, as identified by the green text balloon in the figure), the result of a programming error in the client's asynchronous request queue.

Transaction bottlenecks

The overall performance of a SOA application depends upon many independent factors, such as

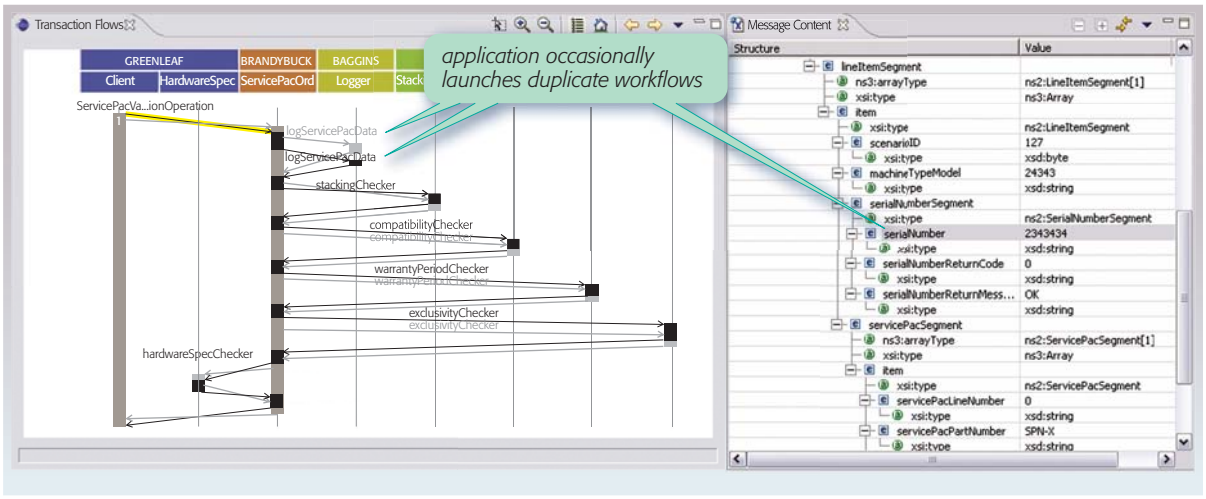


Figure 7
Detecting the occasional creation of duplicate requests in the order validation service

the bandwidth of the network, the responsiveness of the services it invokes, the efficiency of the application itself, and the overhead of the middleware that supports it. When an application fails to perform as expected, sorting out which of these factors is responsible can be difficult.

The Transaction Flows view of an early version of the online order validation service discussed previously shows where time is spent during a particular invocation of the workflow engine, as illustrated in ***Figure 8***. The shading of the vertical bar represents processing for the ServicePacOrderValidation service (the darker shade for "working," the lighter shade for "waiting," as identified by the upper text balloon in the figure). The Transaction Flows view shows that the responsiveness of two services was a contributing factor, although the time measurements revealed in the tooltip windows (identified by the lower text balloon in the figure) indicate that it was not the dominant factor (about 0.35 seconds for one service and 0.26 seconds for the other). The majority of the total transaction time of 1.75 seconds was spent in the workflow engine.

The Transaction Flow and Statistics Table views of the customer information search application, illustrated in Figure 3, showed that it performed as expected. The network delay on the links to the corporate data center (1.364 seconds each to transmit the request and response messages, highlighted in yellow in the figure) and the response time from the data center (2.151 seconds to process the operation on the server) were substantial, but the project manager did not consider these to be serious problems because the Service Topology and Flow Patterns view, illustrated in Figure 2, showed that most searches were satisfied by the local high-performance database.

Unavailable resources

SOA applications depend upon the stability of many geographically and organizationally distributed services. When hardware or software problems make any of those services unavailable or unresponsive, locating that service and handling the situation may be straightforward for shallow topologies of well-understood services. However, when applications fail somewhere within a nested workflow involving many services, or because services with unfamiliar semantics return unexpected responses, simply locating and diagnosing the problem can be daunting.

The customer information search application mentioned earlier failed intermittently, but the tester could not determine the reason from the diagnostics returned to the point-of-sale client. The tester could have tracked down the source of the problem by digging through the server logs on the machines involved in the application, but Web Services Navigator located the problem automatically. ***Figure 9*** shows the Transaction Flows and Statistics Tables views: services that fail to respond to requests are

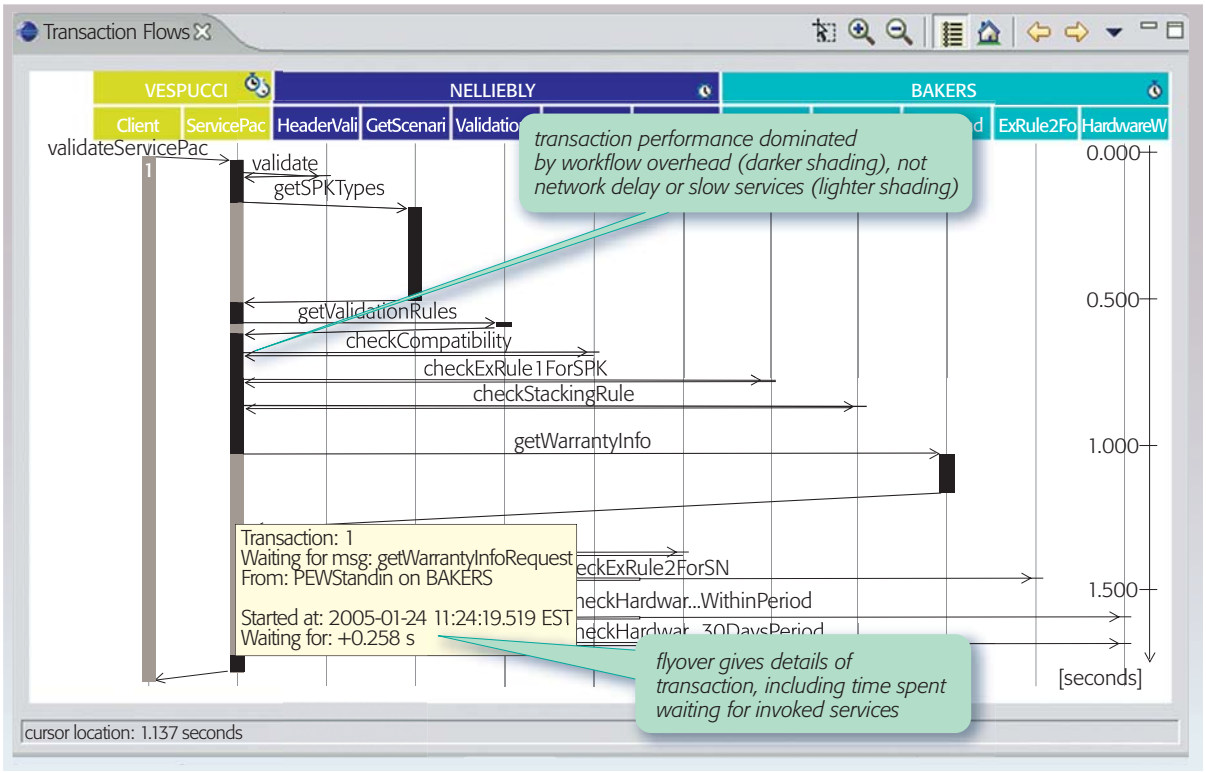


Figure 8
The responsiveness of the order validation service is dominated by the overhead of the workflow engine

flagged; incomplete invocations are identified with question marks ("?") in all views. In this failure, a "getFromThirdPartyDB" request was sent by the EnterpriseCustomer Service but never arrived at its destination.

The online order validation service mentioned earlier also failed intermittently, but Web Services Navigator never showed any incomplete invocations in the failed transactions. However, the linkage between the Transaction Flows and Message Content views allowed a tester to step through a failed transaction, message by message, examining each in succession. *Figure 10* shows a valid request to the CheckCompatibility service (the top Message Content view) but an apparently invalid response (bottom Message Content view) in which the required fields are empty. An ensuing discussion with the programmer revealed that the request to the CheckCompatibility service resulted in a relational database query, which could fail in a variety of ways. Because the service interface did not provide for any explicit error indicators, the service implementation indicated an error by returning an empty response. In this case, the error was that the relational database was offline.

Syntax errors

Web Services encode application data into XML when sending messages and decode XML into application data when receiving messages. The mappings between language-specific data types used within applications and the XML data types used in SOAP messages conceal many opportunities for syntax and semantic errors. Moreover, the details of type mapping have evolved as Web Services technology has matured, resulting in subtle incompatibilities between successive versions of SOAP engines.

A customer information update application was developed to support the retail point-of-sale system mentioned earlier. The application was designed to synchronously update all of the enterprise's databases when customers register product warranties. As testing progressed, records were found that had not been correctly updated. The linkage between the Transaction Flows and Message Content views allowed a developer to step through an apparently

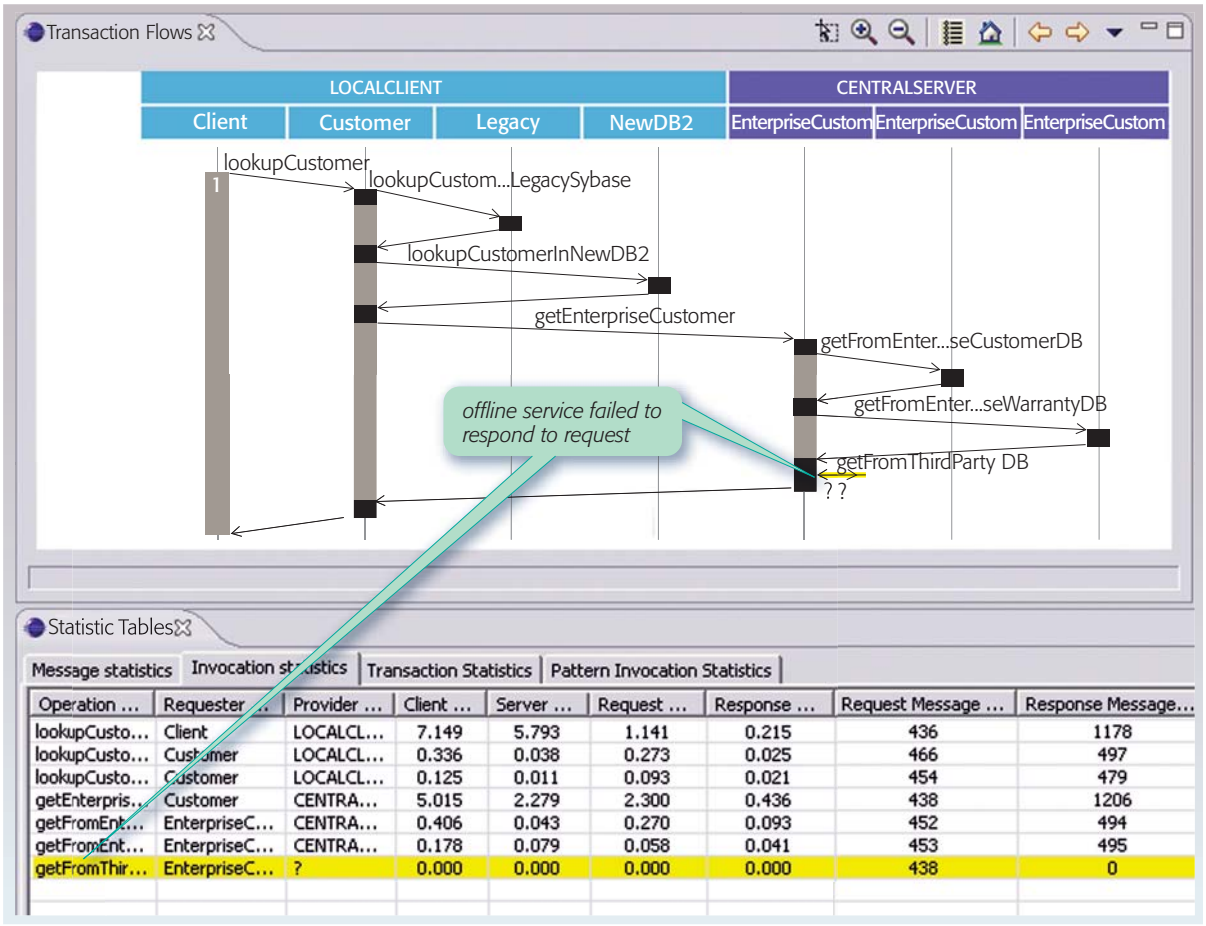


Figure 9
Diagnosing a failure to respond to requests in the retail point-of-sale system

successful transaction, message by message, examining each in succession. ***Figure 11*** showed the developer that in some but not all messages, XML elements with no value had the "nil" attribute.

Further testing revealed that some SOAP engines correctly encoded null Java strings into XML elements with the "nil" attribute, but other SOAP engines incorrectly omitted the "nil" attribute. All of the SOAP engines correctly decoded "nil" XML elements into null Java strings, and correctly decoded "non-nil" XML elements into empty Java strings (that is, non-null Java strings with a length of zero).

The design for the update application specified that a null Java string value meant "leave the field unchanged," whereas an empty Java string value meant "clear the field." Hence, the encoding error in some SOAP engines effectively changed the meaning of updates with null Java strings from "leave this field unchanged" to "clear this field," resulting in incorrect database updates.

The online order validation service mentioned previously was developed and tested with the latest Web Services technology available at the beginning of the project. Later, when it was moved to different versions of middleware, several compatibility problems surfaced. For example, the application represented dates internally with the Java type `GregorianCalendar`. Older versions of the middleware encoded the Java type `GregorianCalendar` as the XML type `date`. Newer versions of the middleware decoded the XML type `date` as the Java type `Calendar`, which caused receivers to return "argument type mismatch" faults. ***Figure 12*** shows the Transaction Flows view and the associated Message

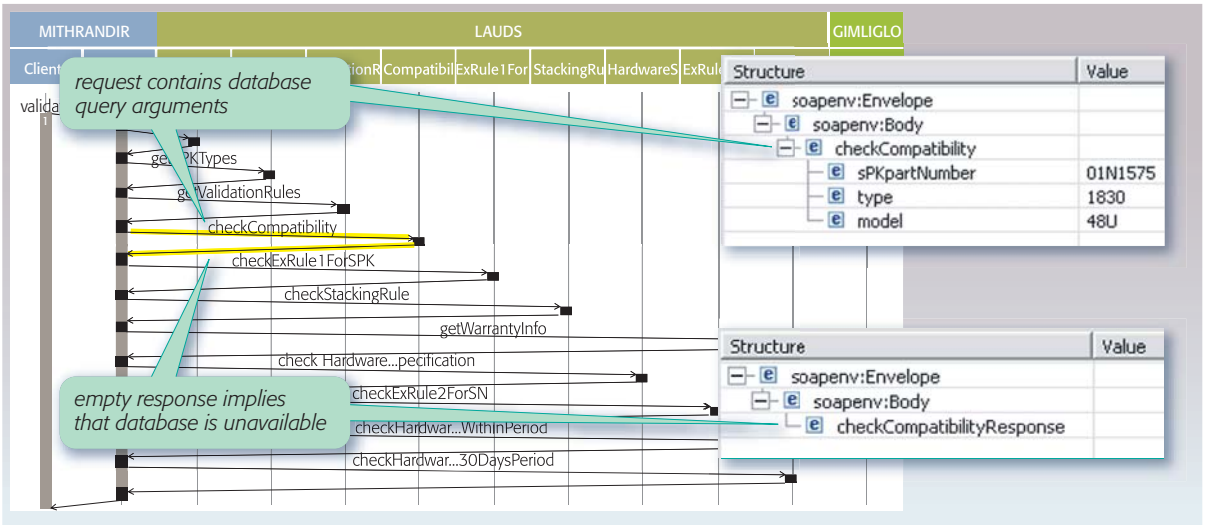


Figure 10
Diagnosing a failure to respond to requests in the order validation service

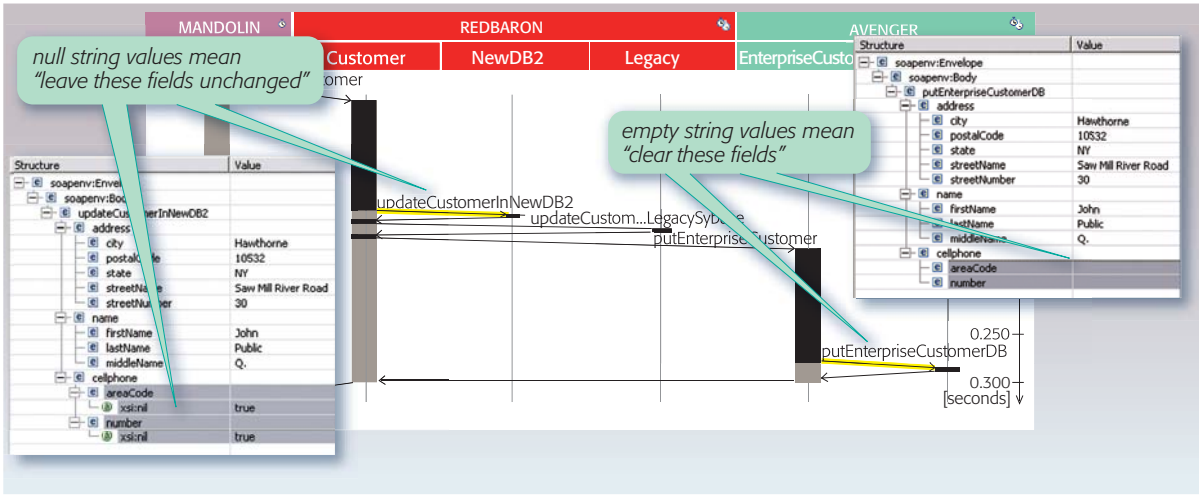


Figure 11
Detecting discrepancies in the ways a NULL Java string is encoded by different SOAP engines

Content view of this error. Newer middleware encoded the Java type `GregorianCalendar` as the XML type `dateTime` instead of `date`, but did not encode a time value along with the date value, causing receivers with older middleware to throw a Java `NumberFormatException`.

DATA COLLECTION

Web Services Navigator uses logs created by the Data Collector for IBM Web Services Navigator. When the Data Collector is installed and configured in WebSphere Application Server, it logs the content and context of SOAP messages[15] sent and received by configured applications. The logs from all machines involved in a Web Services-based application can then be imported into Web Services Navigator. Web Services Navigator automatically reconstructs Web Services transaction flows by correlating send and receive events, and compares transaction flows to identify recurring patterns. The

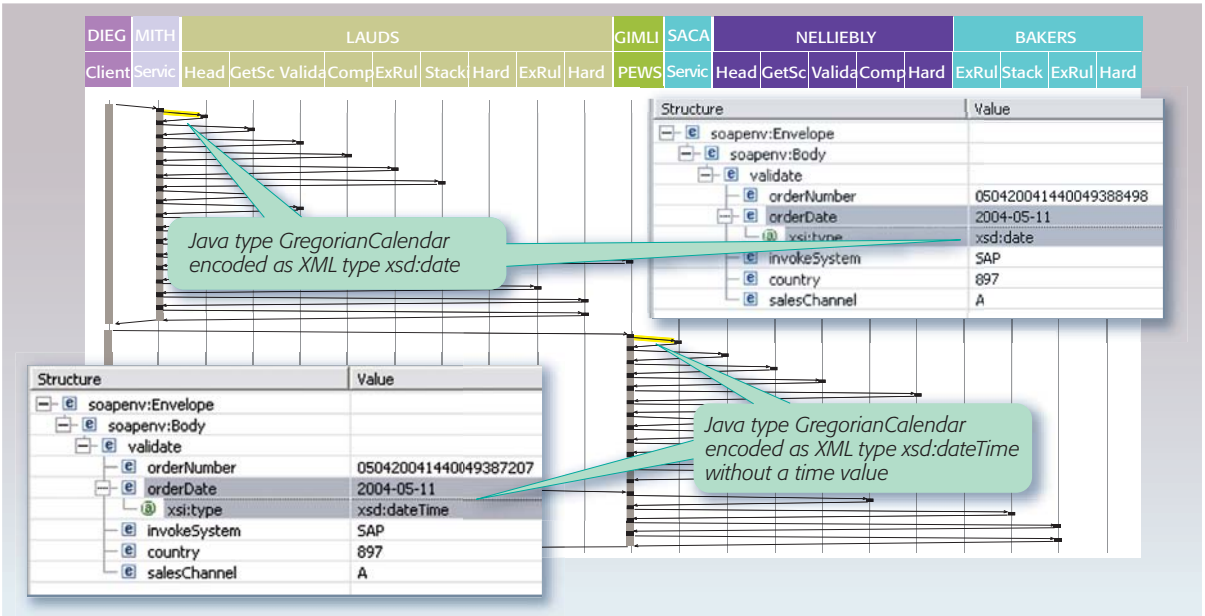


Figure 12
Using Transaction Flows view and Message Content view to determine an incompatibility between different versions of middleware

messages, flows, and patterns can be explored through the Web Services Navigator's five interactive views.

To successfully correlate events and construct message, invocation, transaction, and pattern information, Web Services Navigator needs detailed information about the message content, context (event type, operation name, etc.), and transport (time of event, IP address, etc.) of each event logged.

This section describes the types of log data collected for Web Services Navigator. This paper does not address the transport of collected data, nor the security issues involved in collecting, transporting, and visualizing this data. Data collection is discussed in more detail in "Management of the service-oriented-architecture life cycle" by Cox and Kreger in this issue.[16]

For each event, Web Services Navigator needs the *message content*. This is the SOAP Envelope XML element, encoded as a UTF-8 byte array (UTF-8 is a Unicode** Transformation Format), which contains the SOAP header elements, if any, and the SOAP Body element, which contains the application data.

The Data Collector inserts a proprietary SOAP header element into messages when it logs send events so that Web Services Navigator can correlate them with matching receive events. The proprietary header contains a "correlator" that is similar to an ARM4 correlator.[17] Web Services Navigator makes the application data in SOAP body elements available for inspection in the Message Content view.

For each event, Web Services Navigator needs some information about the *message context*:

- The event type, which is one of the following:
  - — Client request (a request message is sent by a client to a service endpoint)
  - — Server enter (a request message is received by a service endpoint from a client)
  - — Server leave (a response message is returned to a client by a service endpoint)
  - — Server fault (a fault message, instead of a response message, is returned to a client by a service endpoint)
  - — Client response (a response or fault message is received by a client from a service endpoint)

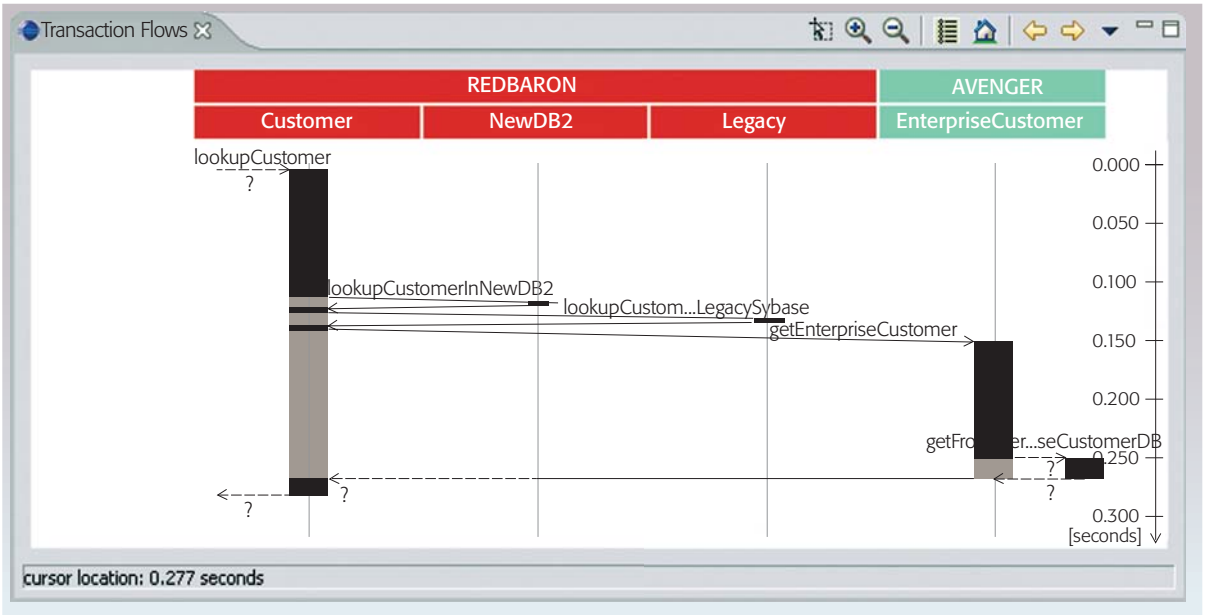


Figure 13
Missing events and inferred messages in the Transaction Flows view

- The operation name (from the WSDL [Web Services Description Language] definition[18] of the service endpoint)
- The operation direction, which is one of the following:
  - Two-way (that is, synchronous request and response messages are expected)
  - One-way (that is, asynchronous request message only, no response message is expected)
  - Unknown (the operation direction is not known to the SOAP engine)
- The service endpoint address (the Uniform Resource Identifier or URI[19] of the service to which request messages are sent)
- The process and thread identifiers under which the SOAP engine is executing

For each event, Web Services Navigator also needs some information about the *message transport*:

- The time of the event, according to the local system clock, with the best precision available
- The local IP name, address, and port number
- The remote IP name, address, and port number
- The HTTP headers preceding the message content

In many situations, Web Services Navigator can infer missing information from redundant data that is logged by the Data Collector and reconstruct transactions even when the data collected is incomplete. This is helpful when the log information is incomplete, for example, when some information for some events is unknown or unavailable, some events on some machines are not logged, or some machines involved in a Web Services-based application are not instrumented. ***Figure 13*** marks missing events with question marks and indicates inferred messages as arrows with dashed lines. On the left side and bottom of this figure, the two question marks near dashed lines suggest that an uninstrumented client called the Customer service. Similarly, no receive event was logged by machine REDBARON for the response message from the EnterpriseCustomer service, but Web Services Navigator was able to infer it from other events logged before and after on the machines involved.

ARCHITECTURE AND IMPLEMENTATION

Web Services Navigator is implemented as a plug-in feature for the Eclipse Workbench Version 3.0. The

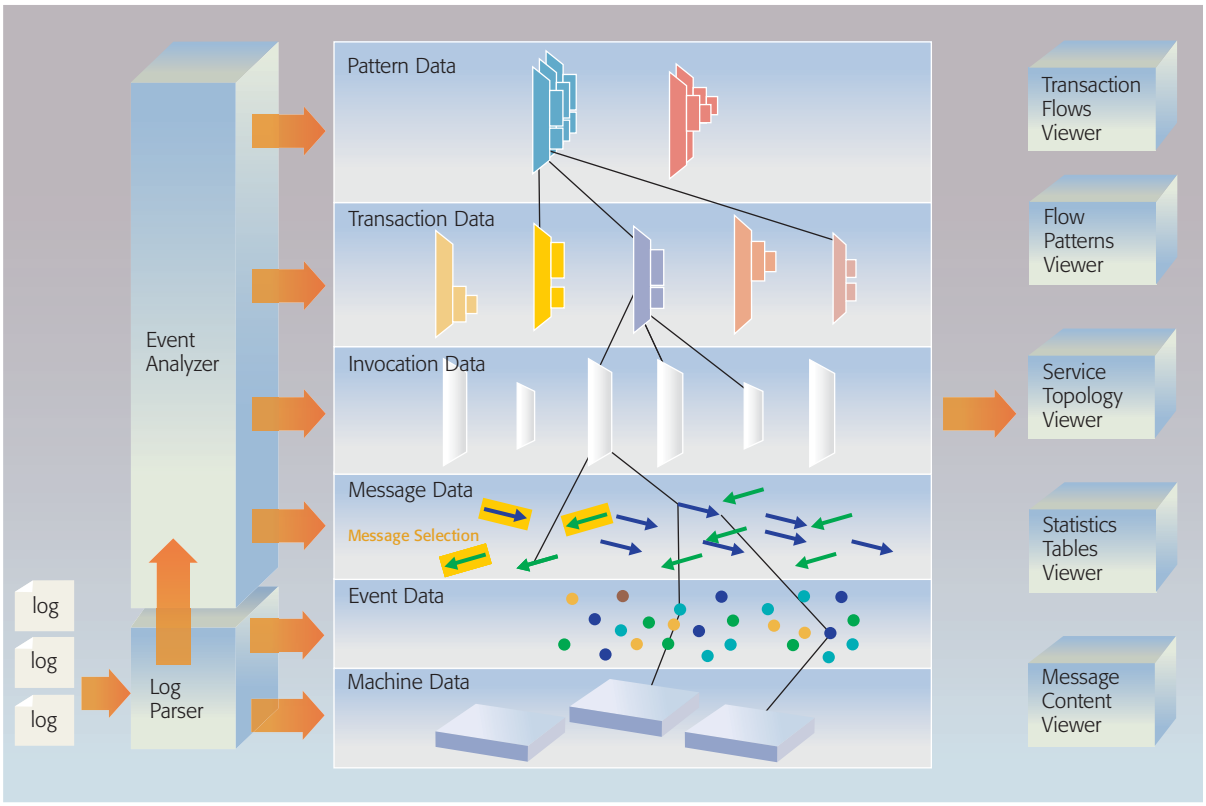


Figure 14
The Web Services Navigator architecture

architecture of Web Services Navigator is illustrated in ***Figure 14***.

The *log parser* reads logs recorded by the Data Collector and constructs a set of events representing the SOAP messages sent and received by the instrumented machines. This information is used by the event analyzer.

The *event analyzer* adds successive layers of structure using the Eclipse Modeling Framework[20] on top of the basic event and machine information, representing messages, invocations, transaction flows, and flow patterns. This information is used by the interactive views.

The *interactive viewers* permit users to explore their applications by examining the message, invocation, transaction flow, and flow pattern information from five distinct and complementary perspectives, the five views of the tool. The five interactive views are linked together: when messages are selected in any view, they are highlighted in all views.

When the data for a Web Services-based application is processed, the logs from the machines involved are imported into Web Services Navigator, which automatically analyzes the events, reconstructs the transaction flows, and identifies flow patterns. The resulting data model of this execution information can then be explored through five interactive views.

To reconstruct Web Services transaction flows and identify flow patterns, Web Services Navigator correlates the logged events and adds several layers of structure on top of them, as illustrated in ***Figure 15***:

1. Matching send events and receive events are combined into messages,
2. Corresponding request and response messages are paired into operation invocations (analogous to remote procedure calls),
3. Related incoming and outgoing invocations are connected into transaction flows (analogous to "call trees"), and

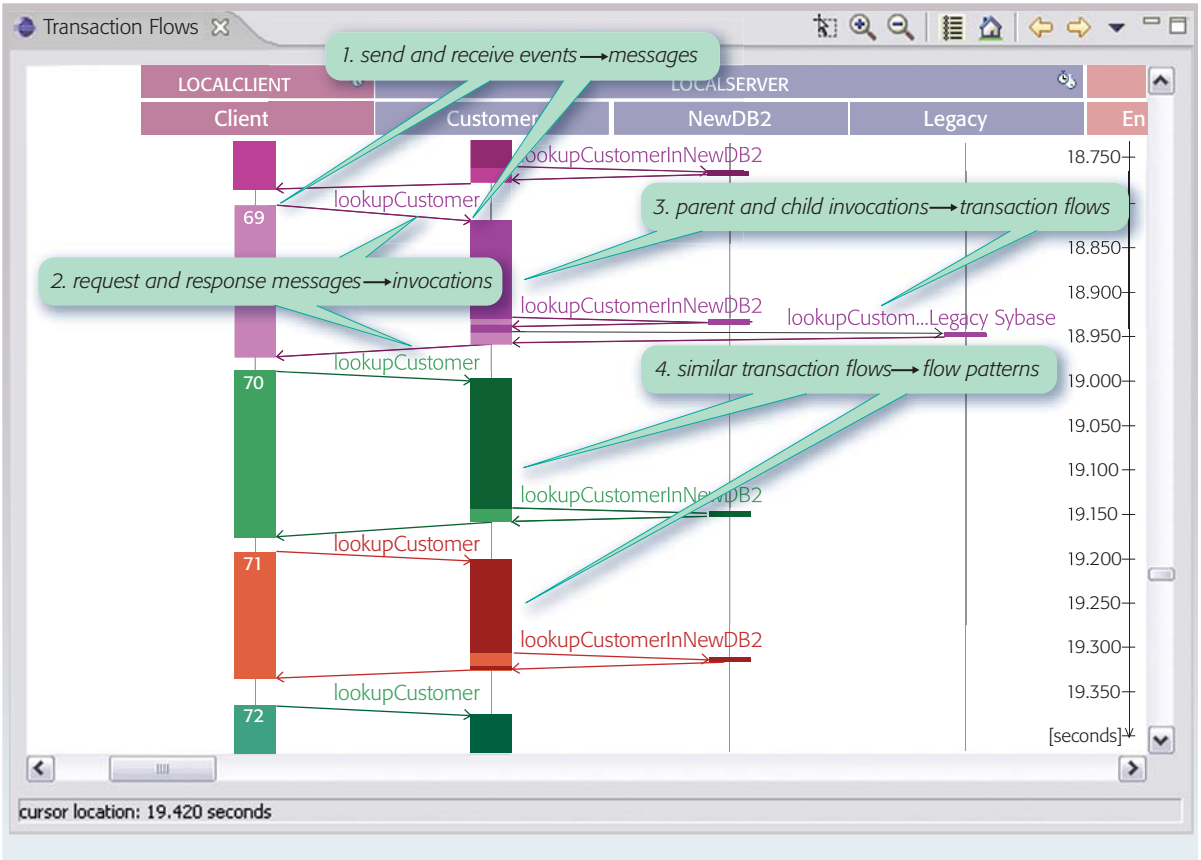


Figure 15
Correlating events into messages, invocations, transactions, and patterns

4. Similar transaction flows are grouped into flow patterns.

Compensating for clock skew

The machines involved in a Web Services-based application may synchronize their system clocks, for example by using the Network Time Protocol (NTP) daemon.[21] However, if system clocks are not synchronized, the time stamps logged for near-simultaneous events may differ by seconds or hours. In the Transaction Flows view, this may cause messages to appear to either jump far ahead or flow backwards in time.

Because no information about clock skew is available in logs imported into Web Services Navigator, the tool correlates events without using time stamps. Then, upper and lower bounds on clock skew are estimated by comparing the send and receive time stamps on messages exchanged by different machines and by applying a modified version of the Floyd-Warshall "all-pairs shortest-path" algorithm.[22] The approximate bounds on clock skew are used in the Transaction Flows view to avoid drawing messages that appear to flow backward in time.

Laying out topology graphs

In the Services Topology view, each node corresponds to a service, and each edge corresponds to an invocation of a service. A hierarchical-cluster-graph layout is used to highlight their structure. Topology graphs are particularly useful for validating actual application execution against the original architectural design. For example, Figure 4 and the left side of Figure 2 show some simple service configurations. The upper right side of Figure 1 illustrates a topology with more complex service relationships; it emphasizes the structure of the application by grouping related services together while minimizing the number of edge bends and crossings. Details of

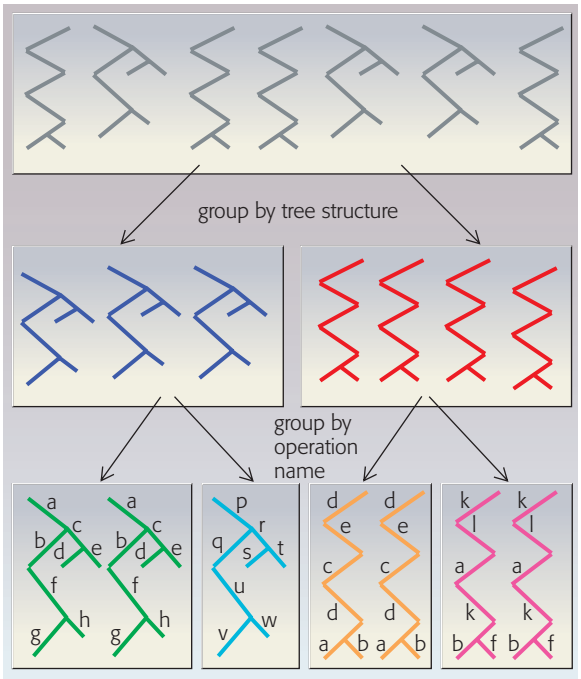


Figure 16
Pattern extraction algorithm

service usage, such as the number of invocations and faults, the total network delay, and the total message load, can be displayed in tooltip windows by moving the cursor over each service.

Our layout algorithm is based on the classical Sugiyama scheme.[23] It is composed of three phases applied to topology graphs to obtain a visually pleasant layout. Each phase addresses a different minimization problem:

1. Assign hierarchical ranks (in our Services Topology view, a rank is the same as a column number, starting from the left) to the nodes in a way that minimizes the number of ranks as well as the number of backward edges.
2. Compute an ordering of nodes within a rank that minimizes the number of edge crossings.
3. Fine-tune the relative position of nodes to minimize the length of edges, as well as the number of edge bends.

One challenge in the design of the layout algorithm came from our specific need to group nodes by service. We thus use a hierarchical cluster graph instead of the hierarchical graph commonly described in the literature.[24]

Another challenge is that the second phase (edge-crossing minimization) of our algorithm is an NP-hard problem. (NP-hard, or Non-deterministic polynomial-time hard, is a concept in computational complexity theory.) We approximate the optimal solution by calculating attractive and repulsive forces on the edges to order the nodes. The third phase (edge-length reduction) also has an extra priority—keeping long edges that span more than two consecutive ranks straight. We first run a long-edge straightening algorithm and then reduce edge length.

The key to the layout algorithm is the use of a combination of attractive and repulsive forces on the edges. These forces directly sort clusters of nodes in the second phase and directly find the positions for clusters in the third phase. It addresses the problem of "clusters," is simple to implement, and yields good visual results (that is, it yields fewer edge crossings and bends, and a more balanced layout). Its performance is comparable to that of the best in the literature for hierarchical graphs:

- For the second phase (ordering the nodes within a rank), the performance of our layout algorithm is $O(m_i + n_i \log n_i)$, where n_i is the number of nodes on the changeable rank i and m_i is the number of edges between two adjacent ranks.
- For the third phase, the layout algorithm is linear. Specifically, its performance is $O(n_i + m_i)$, where n_i is the number of nodes on the changeable rank i and m_i is the number of edges between two adjacent ranks.

This means that our algorithm will perform reasonably well, even with a large number of nodes.

Identifying transaction patterns

Web Services Navigator categorizes transaction flows based on similar patterns of invoked services. Transactions typically start with a root invocation (analogous to a function call), and proceed through one or more service invocations (analogous to a function call tree), possibly branching off via one-way messages into parallel service invocations (analogous to concurrent threads). The algorithm for extracting flow patterns from transaction flows, illustrated in *Figure 16*, involves these steps:

- All of the transaction trees are compared, starting at their roots.

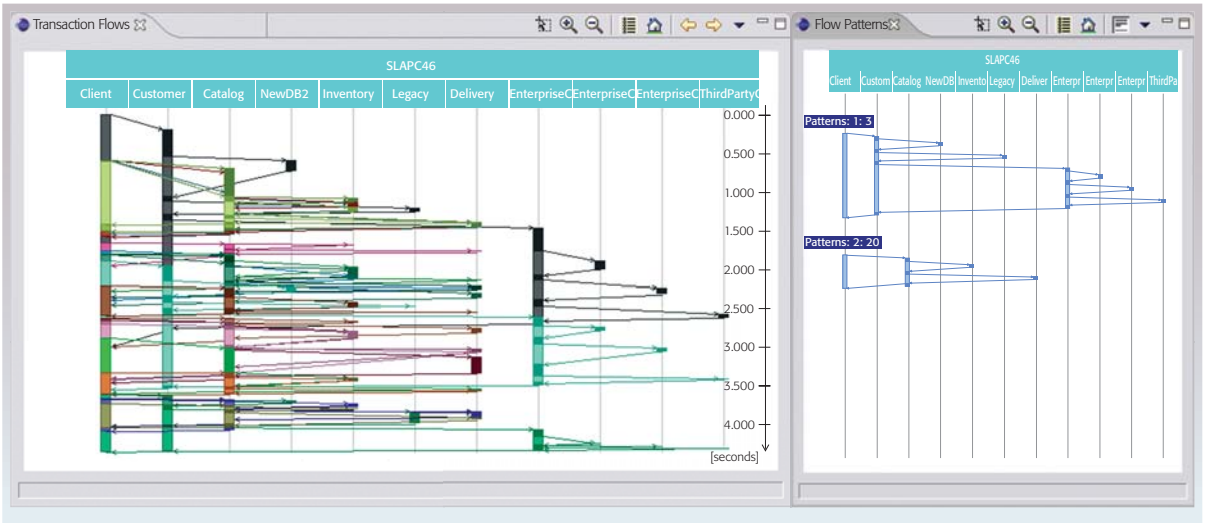


Figure 17
The Flow Patterns view (on the right) reveals patterns in the Transaction Flows view (above)

- All transactions are partitioned into groups of isomorphic invocation trees.
- All isomorphic invocation trees with the same operation names and service points are grouped together.
- A flow pattern is created for each such group of similarly named isomorphic transaction flows.
- Aggregate statistics are calculated for each flow pattern, including minimum, maximum, and average message sizes and response times.

Web Services Navigator uses exact matching for invocation graphs, operation names, and service points, but other categorization algorithms are possible and may be equally useful in some situations. For example, a "fuzzy" matching algorithm might group together transactions with operation names that differ only in case, or a "cluster" matching algorithm might group together transactions that invoke the same operations on different service points.

Pattern extraction fosters understanding by revealing information that is not apparent from more literal visualizations, as illustrated in ***Figure 17***. The Transaction Flows view (on the left in the figure) of a set of transactions shows that they executed concurrently, but it is not apparent that they have anything in common besides the services they invoke. The Flow Patterns view (on the right in the figure), however, reveals that there are two types of transactions, and most of the transactions follow the simpler of the two patterns.

RELATED WORK

Aguilera et al.[25] propose a method for finding performance bottlenecks in distributed systems without installing new tracing agents. They extract message-level traces from the standard logging features on each node, and then use two heuristic algorithms to extract cause and effect relationships. The "nesting" algorithm uses the nested nature of RPC-style (RPC stands for remote procedure call) communication, while the "convolution" algorithm is based on signal-processing techniques. The advantage of this system is that it works on "black boxes," requiring no special tracing or instrumentation. In Web Services Navigator, we try to avoid heuristics by using comprehensive data collection because in complex systems, a tool's effectiveness for problem determination would be compromised by the "false positives and negatives" inherent in a heuristic approach.

Moe and Carr[26] describe a system that traces the execution of Common Object Request Broker Architecture (CORBA**). After intercepting the CORBA calls, they pair up the RPC call-return sequences and aggregate the information into summary statistics, which are presented in a scatter plot. The focus of this system is on exceptions (at the CORBA level) that may occur in a distributed system. The system does not provide a view of

execution at the transaction level and cannot isolate specific messages, making it unsuitable for problem determination.

Paragraph[27] is a tool that shows the message passing of parallel programs on multiprocessors. It has different views that include processor utilization, state of the processors, and logical connectivity of the multiprocessors. Pablo[28] is similar to Paragraph in visualizing the execution of parallel systems. It has additional features like adaptive tracing and shows the execution in 3-D scatter plots. PVaniM[29] helps in optimizing programs running on Parallel Virtual Machines. Its views show aggregate information about messages and processor states. The Paragraph, Pablo, and PVaniM frameworks do not include the correlation information necessary for reconstructing Web Services transactions, and the amount of message-layer detail presented would probably obfuscate the business-layer rules they implement.

ETE[30] shows the itinerary of a single transaction as it goes through different nodes. This system provides a breakdown of the time spent in different components for one transaction.

Visualization has proven to be very helpful in understanding, debugging, and profiling software systems. A lot of work has been done on visualizing how a program executes on one computer.[31–42] The challenges here are mainly related to making a program run faster and with less memory. The complexity of such programs is mainly related to the large amount of data such execution logs can produce. However, with most of this activity happening on one computer, it is relatively easy to collect and correlate the data.

Static analysis is a common technique used to discover problems in software systems. For Web Services in particular, Fu et al.[43] have proposed a generic framework for analyzing BPEL[44] programs by transforming them into Promela[45] programs, a verification language. A limitation of this framework is that it does not handle correlation of process instantiations. A dynamic approach, such as Web Services Navigator, is required for such analysis (in general, static analysis may produce so many possible outcomes that it is impractical for problem determination). Although dynamic analysis can produce very precise results, it may not be exhaustive. The dynamic analysis and visualization of Web Services-based application execution described in this paper is a natural adjunct to static analysis, program-execution tracing and debugging, and code path coverage by means of automated testing. All of these techniques are complementary and can be used together to enhance the quality and performance of Web Services-based applications.

A NEW METHODOLOGY FOR PROBLEM DETERMINATION IN WEB SERVICES

The service-oriented environment poses new challenges for understanding application behavior and for problem determination. Web Services Navigator, a new tool for visualizing the execution of Web Services-based applications, was designed to address these challenges. The tool is the result of a research project that is available on the IBM alphaWorks Web site. In this paper we demonstrated the benefits of the tool by showing a number of problems that have been solved by using the tool in practice, both within IBM and in customer engagements.

The problem cases described in this paper were encountered by Web Services-based application developers in practical situations. The developers using the tool did not undergo extensive training—their only exposure was a short demonstration session. Their experience shows that visualizing the execution of Web Services provides useful insight for problem determination and performance optimization. They used the tool during the development and prototyping phases of their projects and found the Service Topology and the Flow Patterns views especially helpful in discussions with their customers.

The Web Services Navigator sheds new and valuable light on several issues:

- The visual vocabulary of Web Services Navigator helps to bridge conceptual gaps between business and IT specialists, who typically speak different languages. It employs novel algorithms for abstraction and pattern extraction to render Web Services-based application execution logs in ways that are meaningful to audiences including business owners, application developers, and operations staff.
- The five linked views produced by Web Services Navigator cut through the horizontal complexity

that can obscure the essence of Web Services-based application execution. They dramatically simplify such essential tasks as verifying the correctness of a business process implementation or finding the location of process failures.
- The ability of Web Services Navigator to visualize the behavior of applications, independent of programming language and operating system and without modifying those applications, demonstrates the value of the open standards employed by Web Services, such as XML and SOAP.
- Architects typically use descriptive service names that are meaningful to business owners, application developers, and operations staff. Those same names appear in the views displayed by the tool; this facilitates communication between business and technical users.

ACKNOWLEDGMENTS

We have benefited from the enthusiastic cooperation of the eServicePac Validation team, including John Hicks, Sophia Krasikov, Raju Pavuluri, Robert Hoch, and Senthil Velayudham. We are also grateful for the support of the IBM Tivoli* "tiger" team, including John Harter, Kevin Dunphy, David Cox, Sudhakar Chellam, Phil Fritz, Samuel Spiro, Ann Marie Gallagher, Peter Wassel, and Chris O'Conner.

We appreciate the many fruitful discussions with our colleagues in IBM Rational*, including Eric Labadie, Richard Duggan, Vincent Encontre, Sergio Lucio, Simon Johnston, and Harm Sluiman; John Knutson in IBM Tivoli; our colleagues in IBM Global Services, including Olaf Zimmerman and Yaroslav Dunchych; our colleagues in the IBM CIO office, including Lance Walker and Michael Martine; and our colleagues in IBM Research, including Claudia McGhee, Aaron Kershenbaum, and Ian Whalley.

CITED REFERENCES AND NOTES

1. *Web Services Architecture Specification*, World Wide Web Consortium (W3C), http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.
2. K. Gottschalk, S. Graham, H. Kreger, and J. Snell, "Introduction to Web Services Architecture," *IBM Systems Journal* **41**, No. 2, 170–177 (2002).
3. "Boeing 747-400, by the Numbers," Boeing Corporation, http://www.boeing.com/news/feature/747evolution/747facts.html.
4. "Web Services Navigator," alphaWorks Technology, IBM Corporation, http://www.alphaworks.ibm.com/tech/wsnavigator/.
5. "Data Collector for Web Services Navigator," alphaWorks Technology, IBM Corporation, http://alphaworks.ibm.com/tech/wsdatacollector/.
6. IBM Tivoli Composite Application Manager for SOA, Version 6.0, Product Code 5724-MO7, IBM Corporation.
7. IBM Tivoli Monitoring for Web Services, Version 1.1.0, Product Code 5799-GZR, IBM Corporation.
8. "Eclipse Platform," Eclipse Foundation, http://eclipse.org/eclipse/index.html.
9. "Java Development Tools," Eclipse Foundation, http://www.eclipse.org/jdt/index.html.
10. "Test & Performance Tools Platform (TPTP)," Eclipse Foundation, http://eclipse.org/tptp/index.html.
11. "IBM Rational Application Developer" and "IBM Rational Web Developer," IBM Corporation, http://www-306.ibm.com/software/info/developer/radrwd/index.jsp.
12. W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and and J. F. Morar, "Visualizing the Execution of Web Services," IBM alphaWorks, http://www.alphaworks.ibm.com/g/g.nsf/img/semanticsdocs/$file/visualizews.pdf.
13. "Supply Chain Management Sample Application Architecture," Web Services Interoperability Organization, http://www.ws-i.org/SampleApplications/SupplyChainManagement/2003-12/SCMArchitecture1.01.pdf.
14. See Reference 13, Figure 1 on page 6.
15. *Simple Object Access Protocol (SOAP) Version 1.1 Specification*, World Wide Web Consortium (W3C), http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
16. D. Cox and and H. Kreger, "Management of the Service-Oriented-Architecture Life Cycle," *IBM Systems Journal* **44**, No. 4, 709–726 (2005, this issue).
17. *Application Request Measurement (ARM) 4.0 Specification,* The Open Group, http://www.opengroup.org/management/arm.htm/.
18. *Web Services Definition Language (WSDL) Version 1.1 Specification*, World Wide Web Consortium (W3C), http://www.w3.org/TR/wsdl.
19. *Uniform Resource Identifiers (URI) Generic Syntax Specification*, Internet Engineering Task Force (IETF), http://www.ietf.org/rfc/rfc2396.txt.
20. "Eclipse Modeling Framework," Eclipse Foundation, http://www.eclipse.org/emf/.
21. "Network Time Protocol," the *Network Time Synchronization Project*, http://www.ntp.org/.
22. T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press and McGraw-Hill (1990), pp. 560–562.
23. K. Sugiyama, S. Tagawa, and M. Toda, "Methods for Visual Understanding of Hierarchical Systems," *IEEE Transactions on Systems, Man and Cybernetics,* Volume SMC-11, No. 2, 109–125 (1981).
24. G. di Battista, P. Eades, R. Tamassia, and I. Tollis, *Graph Drawing, Algorithms for the Visualization of Graphs*, Prentice Hall, Upper Saddle River, New Jersey (1999).
25. M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Distributed Systems of Black Boxes," *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP'03)*, (October 2003), pp. 74–89.

26. J. Moe and D. A. Carr, "Understanding Distributed Systems Via Execution Trace Data," *International Workshop on Program Comprehension,* IEEE Computer Society Press, New York (2001), pp. 60–69.

27. M. T. Heath and J. A. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software* **8**, Issue 5, 29–39 (September 1991).

28. D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment," *Proceedings of the Scalable Parallel Libraries Conference,* IEEE Computer Society (1993), pp. 104–113.

29. B. Topol, J. T. Stasko, and V. Sunderam, "PVaniM: A Tool for Visualization in Network Computing Environments," *Concurrency: Practice and Experience* **10**, No. 14, 1197–1222 (1998).

30. J. L. Hellerstein, M. Maccabee, W. N. Mills, and J. J. Turek, "ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems," *International Conference on Distributed Computing Systems* (1999), pp. 152–162.

31. D. Jerding and J. T. Stasko, "The Information Mural: A Technique for Displaying and Navigating Large Information Spaces," *Proceedings of the IEEE Symposium on Information Visualization,* Atlanta, GA (October 1995), pp. 43–50.

32. D. Jerding, J. T. Stasko, and T. Ball, "Visualizing Interactions in Program Executions," *Proceedings of the 19th International Conference on Software Engineering* (May 1997), pp. 360–370.

33. D. Kimelman, B. Rosenburg, and T. Roth, "Visualization of Dynamics in Real World Software Systems," *Software Visualization: Programming as a Multimedia Experience*, J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, Editors, MIT Press (1998), pp. 293–314.

34. E. Kraemer, "Visualizing Concurrent Programs," *Software Visualization: Programming as a Multimedia Experience*, J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, Editors, MIT Press (1998), pp. 237–256.

35. W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, "Visualizing the Execution of Java Programs," *Proceedings of the International Seminar on Software Visualization,* S. Diehl, Editor, Volume 2269 in Lecture Notes in Computer Science, Springer-Verlag, New York (2001), pp. 151–162.

36. W. De Pauw and and Gary Sevitsky, "Visualizing Reference Patterns for Solving Memory Leaks in Java," *Proceedings of the ECOOP '99 European Conference on Object-Oriented Programming* (1999), pp. 116–134.

37. W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan, "Drive-by Analysis of Running Programs," *Proceedings of the ICSE Workshop of Software Visualization* (May 2001), pp. 17–22.

38. S. P. Reiss, "Visualization for Software Engineering—Programming Environments," *Software Visualization: Programming as a Multimedia Experience*, J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, Editors, MIT Press (1997), pp. 259–276.

39. S. P. Reiss, "Bee/Hive: a Software Visualization Backend," *IEEE Workshop on Software Visualization* (May 2001), pp. 44–48.

40. S. P. Reiss, "An Overview of BLOOM," Program Analysis for Software Tools and Engineering (PASTE '01) (June 2001), pp. 2–5.

41. S. P. Reiss, "Visualizing Java in Action," *Proceedings of the IEEE International Conference on Software Visualization* (2003), pp. 123–132.

42. S. P. Reiss, "JIVE: Visualizing Java in Action," *Proceedings of International Conference on Software Engineering (ICSE 2003)* (May 2003), pp. 820–821.

43. X. Fu, T. Bultan, and J. Su, "Analysis of Interacting BPEL Web Services," *Proceedings of International WWW Conference,* New York, USA (2004), pp. 621–630.

44. Business Process Execution Language (BPEL) for Web Services Version 1.1 Specification, Organization for the Advancement of Structured Information Standards (OASIS), http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

45. "Promela Language Reference," the Spin Project, http://spinroot.com/spin/Man/index.html.

Accepted for publication May 24, 2005.
Published online October 25, 2005.

Wim De Pauw
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (wim@us.ibm.com). Dr. De Pauw received a Ph.D. degree in computer science from the University of Ghent, Belgium, in 1991. At the Watson Research Center, where he has been a research staff member since 1992, he has led the software visualization projects Ovation (C++ visualization) and Jinsight (Java visualization). Jinsight technology is now part of IBM's WebSphere Studio Application Developer, and parts of it have been released under the open-source project Hyades. Dr. De Pauw's current activities include software visualization, Web services, service-oriented architecture, profiling, and debugging, and he is currently the architect and technical leader of Web Services Navigator. His scientific interests include addressing large horizontal complexities in systems by using visualization, pattern extraction, and adaptive tracing techniques. He has published articles in refereed journals, has chaired conferences, and holds several patents in software visualization, tracing, and debugging.

Michelle Lei
(mlei04@gmail.com). As a software engineer at the Watson Research Center, Ms. Lei worked on the Websight project. She received her Master's degree from the Swiss Federal Institute of Technology (EPFL) in March 2004. Her areas of interest include graph layout algorithms, Web technologies, and middleware. She currently resides in Switzerland and works as Project Manager for Java development at the Office Fédéral d'Informatique et Télécommunication in Geneva.

Edward Pring
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (pring@watson.ibm.com). Mr. Pring, a Senior Programmer at the Watson Research Center, received an M.S. degree in computer science from New York University. He has contributed to a wide range of IBM products and technologies, including operating systems, publishing applications, terminal emulators for mainframes, virus protection for personal computers, network automation for the Digital Immune System, visualization, and performance analysis for Web Services. He holds a patent portfolio that spans all of these fields.

Lionel Villard
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (villard@us.ibm.com).

Dr. Villard, an Advisory Research Engineer at the Watson Research Center, received a Ph.D. degree at the Institut National Polytechnique de Grenoble (INPG) in 2002. His research interests include multimedia documents, contextual adaptation, authoring tools, document transformations, incremental transformations, and high performance.

Matthew Arnold
IBM Research Division, Thomas, J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (marnold@us.ibm.com). Dr. Arnold, a research staff member at the Watson Research Center, received a Ph.D. degree in computer science from Rutgers University in 2002. His thesis focused on profiling and optimization techniques for the Java programming language. His current research interests include software profiling, program understanding, and dynamic optimization.

John F. Morar
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (morar@watson.ibm.com). Dr. Morar received a Ph.D. degree in experimental solid-state physics from the University of Maryland in 1982. After joining IBM, he spent two years in residence at the National Synchrotron Light Source, Brookhaven National Laboratory, where he used soft X-ray spectroscopy to probe the outer few atomic layers of semiconductors. Over the following eight years he worked on developing metastable semiconductors using molecular beam epitaxy. Dr. Morar spent seven years in computer virus research, where he managed the Anti-Virus Technology and Systems group. He contributed to numerous releases of the IBM Anti-Virus and Digital Immune System, which is designed to find, analyze, and automatically distribute the cure for a new computer virus faster than the virus itself can spread. He has written 70 articles in peer-reviewed scientific journals and has contributed to IBM's patent portfolio in the areas of device processing, computer virus detection, Web services, and economic systems. Dr. Morar currently manages a group that focuses on the use of Web services both within and between enterprises. ■