RC23703 (W0508-127) August 23, 2005
Computer Science

IBM Research Report

The Diary of a Datum: An Approach to Analyzing Runtime Complexity in Framework-Based Applications

Nick Mitchell, Gary Sevitsky, Harini Srinivasan
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

The Diary of a Datum: An Approach to Analyzing Runtime Complexity in Framework-Based Applications

Nick Mitchell
IBM TJ Watson Research Center
19 Skyline Drive
Hawthorne, NY USA
+1 914-784-7715
nickm@us.ibm.com

Gary Sevitsky
IBM TJ Watson Research Center
19 Skyline Drive
Hawthorne, NY USA
+1 914-784-7619
sevitsky@us.ibm.com

Harini Srinivasan
IBM TJ Watson Research Center
19 Skyline Drive
Hawthorne, NY USA
+1 914-766-1885
harini@us.ibm.com

ABSTRACT
In large-scale framework-based applications, every piece of information has a complex story to tell about its journey. As it makes its way through a tangle of reusable frameworks, it may be transformed from a string, to an Integer, to an integer, and finally to a date. Over the past several years, our research group has analyzed dozens of framework-based applications. We have found it increasingly difficult to understand behavior, weigh design tradeoffs, and assess if and how performance problems can be fixed. Often, simple functionality requires a seemingly excessive amount of runtime activity and complexity.

Much of this activity revolves around the transformation of information from one form to another. In this paper we present an approach to understanding runtime behavior that structures activity in these terms. We show strategies for grouping and filtering activity into a hierarchy of data flow diagrams representing transformations and the flow between them. The approach focuses the analysis on a user-defined scenario, and structures what otherwise would have been an overwhelming amount of information about a run. Next, we give a detailed example that illustrates this approach, and also demonstrates the complexities typically found in this class of application. Finally, we show how structuring the activity into a hierarchy of data flow diagrams allows us to introduce measures of complexity derived from the topology of the diagrams.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *complexity measures*

General Terms
Measurement, Performance, Design

Keywords
Dynamic analysis, program understanding, complexity assessment, performance analysis, design recovery

1.INTRODUCTION
Large-scale applications are being built from increasingly many reusable frameworks, such as web application servers (that include SOAP, EJBs, JSPs), portal servers, client platforms (Eclipse), and industry-specific frameworks. Over the past several years, our research group has analyzed dozens of framework-based applications. We have found that it is becoming more and more difficult to evaluate performance – not only to localize problems, but to assess if and how they can be fixed. These applications may execute enormous amounts of activity to accomplish simple tasks. For example, as shown in Figure 1, a stock brokerage benchmark executes 46971 method calls and creates 7407 objects to convey ten stock holding records from one format to another. How can we tell if the activity is appropriate? If not, is the fault in the application architecture? Or in a particular framework implementation or interface design choice? And why weren't the compilers able to fix it?

Many of these difficulties stem from the nature of framework-based applications. While the implementation of each application is different, we have observed many similarities in their behavior. The most noticeable phenomenon is the amount of effort expended taking information available in one form and transforming it into another. The prevalence of these *transformations* of data seems like a natural consequence of the many standards and legacy systems these systems must integrate. In a web-based server application, the data arrives in one format, is transformed into a Java business object, and is sent to a browser or another system – e.g. from SOAP [5], to an EJB, and finally to XML. More steps are not uncommon. In addition, each step may require work to *facilitate* the transformation, such as looking up a suitable SOAP deserializer.

Many types of processing activity can be viewed as transformations. Some preserve the information content, despite changing the physical form of the data. Some transformations exchange one kind of information for another, such as looking up schema information given a type name; others may affect a change in a value, such as adding sales tax to a subtotal.

Furthermore, it is common for the implementation to be in fact a hierarchy of such transformations. That is, inside any transformation or facilitation, we often find a similar picture, the result of lower-level framework couplings.

Understanding the behavior and costs of a hierarchy of transformations is difficult with existing performance analysis tools. This is partly because problems can rarely be blamed on a single hot method. They are usually caused by a constellation of inexpensive calls that span multiple frameworks. What's more, problems are less likely to be caused by a poor algorithm choice than by the combined design and implementation choices made in disparate frameworks. A single transformation may consist of multiple calls not contiguous in time, and may be implemented by only parts of some methods. In addition, transformations are mediated by data that carry information from one method to another. All of this means that solely focusing on static artifacts, such as methods or components, or on control-oriented abstractions, such as call paths or sequences [2,3,7,8,14,16]

cannot sufficiently capture the behavior of framework-based applications.

This paper introduces an approach to understanding the nature and causes of inefficiencies in these applications. Our approach structures run-time activity as a hierarchy of data flow diagrams representing transformations of information. The approach we present is currently a manual approach. Certain elements of the approach may be amenable to automation; other elements will, by design, require user input.

In Section 2 we describe this approach, consisting of strategies for grouping and filtering the activity, and give a brief example. In Section 3 we give an in-depth example, that follows a single field from a SOAP response into a Java business object – a seemingly simple operation with surprisingly complex behavior. In addition to illustrating the approach, this example illustrates the nature and magnitude of the complexities found in large-scale framework-based applications. Structuring by logical data flow also enables new quantitative analyses that can shed light on the complexity of an implementation. In Section 4 we show some measures derived from topology.

2.STRUCTURING APPROACH

We structure runtime activity as a hierarchy of data flow diagrams. Constructing any such diagram requires strategies for both grouping and filtering the activity. User guidance is required in both of these areas to make the diagrams manageable in size, and useful for a desired analysis task.

There are many ways the same processing activity can be grouped into transformations and diagrams. Choices need to be made about which activity constitutes each transformation, and which transformations to hide in a subdiagram. These choices can be crucial for enabling understanding. In practice we have found two approaches sufficient to cover complex cases, such as that shown in Section 3: grouping based on granularity of information, and by architectural units.

Applications often have semantic notions of *granularity* that cut across multiple type systems. For example, a stock holding record, whether represented in the database or as a Java object, may still be thought of as a record. Our first rule is to group activity so that each diagram shows information flow at a single level of granularity. Note that in some cases it makes sense to include transformations that are transitions between two levels, such as extracting a field from a record.

It is often useful to group activity in a way that makes apparent the party responsible for a given cost. This kind of grouping by *architectural unit* [14,16] can be used to both structure activity into transformations, and to choose which transformations constitute a level of data flow diagram. A common example of the latter would be to distinguish activity the application is responsible for from activity hidden in frameworks it is using. The choice of architectural units is left to the user. We have found that they do not always align with package structure.

No matter how good the grouping strategy, for large-scale framework-based applications, analyzing an entire run would be overwhelming for the user. Users typically have a more limited *analysis scenario* in mind. They commonly want to follow the activity required to produce a specific target datum. Furthermore, they are not interested in seeing every level of detail at once, or every prior contributor to this datum. In our approach, the user defines the following four criteria to specify an analysis scenario:

- The target datum
- The granularity on which to focus
- The starting point, for example a point in time, a starting set of data (e.g. a record that has just been retrieved from a database), or a specific method invocation
- Additional filtering criteria, such as limiting the scenario to the current thread

Figure 1 illustrates one choice of analysis scenario and grouping strategy, on a configuration of the Trade 6 benchmark [10] that acts as a SOAP client.[1] Our analysis scenario is a web transaction that returns a user's stock holdings with quotes. It follows the information, at the granularity of a set of records, as it is returned from a remote web service via SOAP and formatted into HTML. We are therefore able to omit the construction of the request, as well as any advance work building data structures that were not specific to this response. We have grouped activity into transformations so that the flow of information is consistently at the granularity of a set of records, since that was the granularity of the target data.

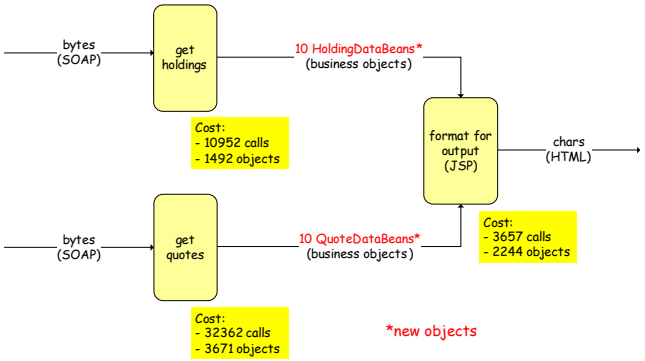

Figure 1. Overview: response to a user request to get holdings.

This figure highlights two of the advantages of organizing an analysis by data flow. First, it unifies disparate activity. The "get quotes" box brings together activity from many different frameworks. Second, the approach helps us relate a cost of some processing activity to the data it produced. We can annotate diagrams with various characteristics of each transformation or data flow. In this figure we show the cost of each transformation in terms of method calls and object creations. The "get holdings" step cost 10952 calls and 1492 new objects, mostly temporaries, to produce the ten HoldingDataBeans from the SOAP response.

We gathered information about the run using a combination of tools, LiveJinsight [7] and ArcFlow [1], and by looking at source code in some cases. The example was run on a publicly available web application server and JVM. The application was warmed up sufficiently to simulate steady-state conditions of a production

[1] We use descriptive labels for boxes, and assume information is available elsewhere about the run-time artifacts they represent. An edge label may show just the top-level object of a data structure. We also omit the standard notation for sources and sinks. Edges left open at one end may terminate in a source or sink, or in some other processing outside the scope of the diagram.

server. The analysis tools report the actual execution, after JIT optimizations were made.

3.THE DIARY OF A DATE

We now walk through a longer example from the same transaction in the Trade benchmark. The example serves to illustrate our approach, and also to illustrate the kinds of complexity that we have seen in real-world framework-based applications. We chose a benchmark example that has been well-tuned at the application level, and therefore demonstrates the challenges in identifying where the limits to further improvement are.

- **Analysis scenario.** We follow how one field of a stock holding is transformed from a portion of a SOAP response message into a field in a Java business object. We chose the purchaseDate field, the most expensive field to process in a HoldingDataBean.
- **Grouping strategy.** We break the scenario into three levels of diagram. In this case we chose to highlight major architectural boundaries (e.g. application vs. framework), along with granularity.

Diagram level 0. Figure 2 shows the top level diagram, containing two transformations. The first takes the bytes representing the date in a SOAP response and builds a Calendar field in a Java business object. This step was a sequence of four calls that were made at the application level. We group them because, together, they perform that function on the purchase date. The second step converts the Calendar to a Date field in another Java business object. We concentrate on the first step.

Diagram level 1. Zooming in on the first step gives us the diagram in Figure 3. The main flow of data is along the middle row of boxes.

The first step extracts bytes from the XML text of a SOAP message, and converts the date and the name of the field we are processing to Strings. The date String is passed to a deserializer for parsing. The SOAP framework allows registration of deserializers for datatypes that can appear in messages. In the lower left corner is a sequence of transformations that look up the appropriate deserializer given the field name String.

We highlight as a group the five transformations related to parsing, to make it easier to see this functional relationship. The first step takes the String, extracts and parses the time zone and milliseconds, and copies the remaining characters into a new String. The reformatted date String is then passed to the SimpleDateFormat library class for parsing. This is an expensive step, creating 38 temporary objects. Zoom level 2 shows why.[2] It returns a new Date object, to which the original time zone and milliseconds are then added back.

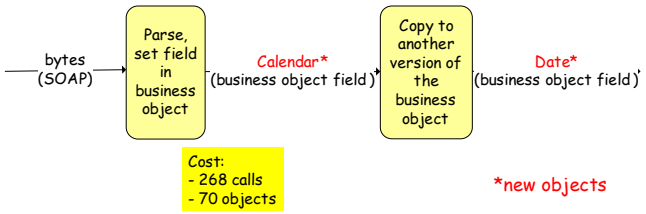

Figure 2. Diary of a Date – Diagram Level 0 (Application)

The Java library has two date classes. A Date object stores the number of milliseconds since a fixed point in time. A Calendar stores a date in two different forms, and can convert between them. One form is the same as in Date; the other is seventeen integer fields that are useful for operating on dates, such as year, month, day, hour, or day of the week.

In the top row is an expensive step that builds a new Calendar. Our Date object is then used to set the value of this Calendar. Finally, that Calendar becomes the purchaseDate field of our business object, via a reflective call to a setter method. Java's reflection interface requires the Calendar to first be packaged into an object array.

Just at this level, the input bytes undergo seven transformations before exiting as a Calendar field in the Java business object.

Diagram level 2. Figure 4 zooms into the SimpleDateFormat parse step. The String containing the date is input, and each of its six subfields – year, month, day, hour, minute, and second – is extracted and parsed individually. Note that although this diagram is at the field granularity, we need to understand how subfields are extracted and recombined to form fields. We therefore choose to show just the top-level transformations at that next lower level of granularity.

The SimpleDateFormat maintains its own Calendar, different from the one discussed earlier at the SOAP level. Once a subfield of date has been extracted and parsed into an integer, the corresponding field of the Calendar is set. After all six subfields are set, the Calendar converts this field representation into a time representation. This is then used to create a new Date object.

Diagram level 3. Figure 5 shows the detail of extracting and parsing a single date subfield, for example, a year. We can see that even at this level, there are six transformations needed to convert a few characters in the String into an integer.

The first five transformations are performed using the general purpose DecimalFormat class from the standard Java library, which can parse or format any kind of decimal number. SimpleDateFormat, however, uses it for a special case, to parse integer months, days, and years. The first, fifth, and sixth transformations are necessary only because of this over-generality. The first transformation looks for a decimal point, an E for scientific notation, and rewraps the characters.[3] Furthermore, since DecimalFormat.parse() returns either a double or long value, the fifth transformation is only needed to box the return value into an Object, and the sixth transformation is only necessary to unbox it.

Discussion. Note that we have made a number of grouping judgements throughout the above example. At the top zoom level, we chose to group parse and set field as a single step. We could have also made these into two separate transformations at the top level. We chose not to because in the SOAP interface there is no way for an application to call them separately. We chose to use our grouping to bring out an architectural feature: our top-level scenario describes the application level, and the next zoom level shows the SOAP framework. In general, the flexibility in grouping allows us to bring out different features of interest for a given analysis.

[2] In our experience, many things named "Simple" are expensive.

[3] It even checks fitsIntoLong() on a number representing a month!

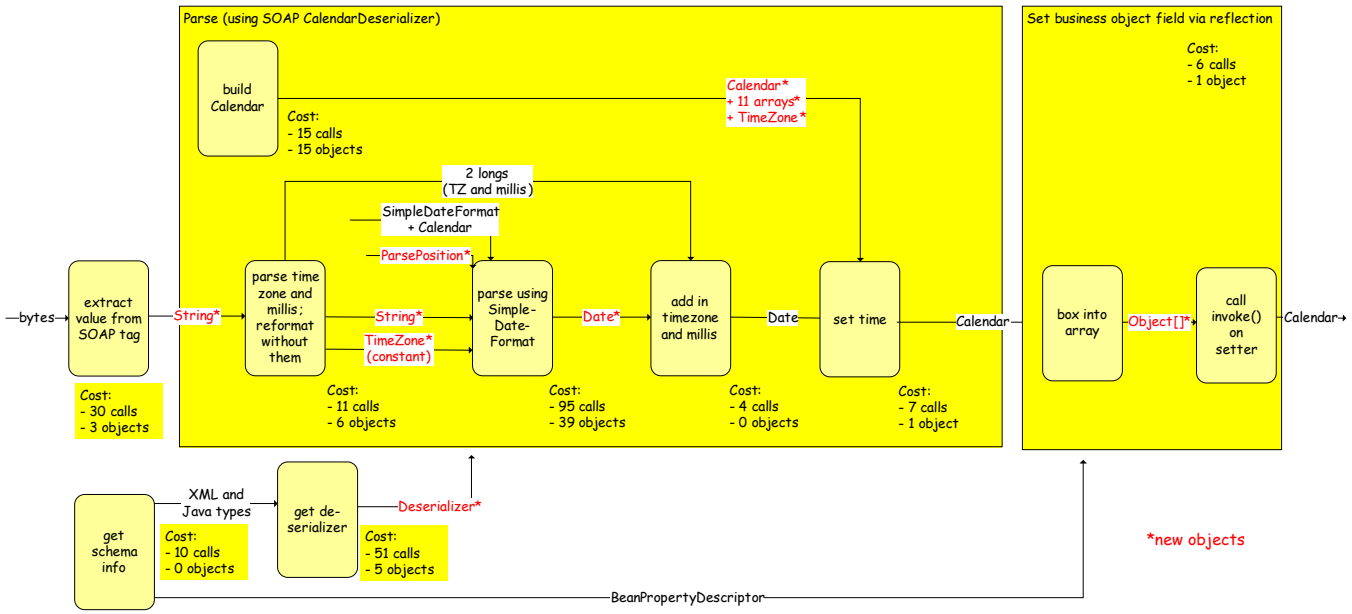

Figure 3. Diary of a Date – Diagram Level 1 (SOAP)

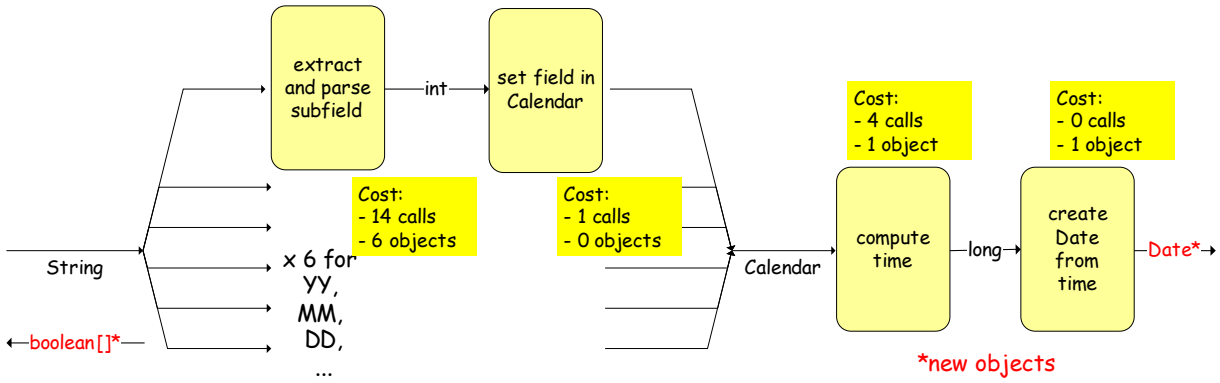

Figure 4. Diary of a Date – Diagram Level 2 (Java library)

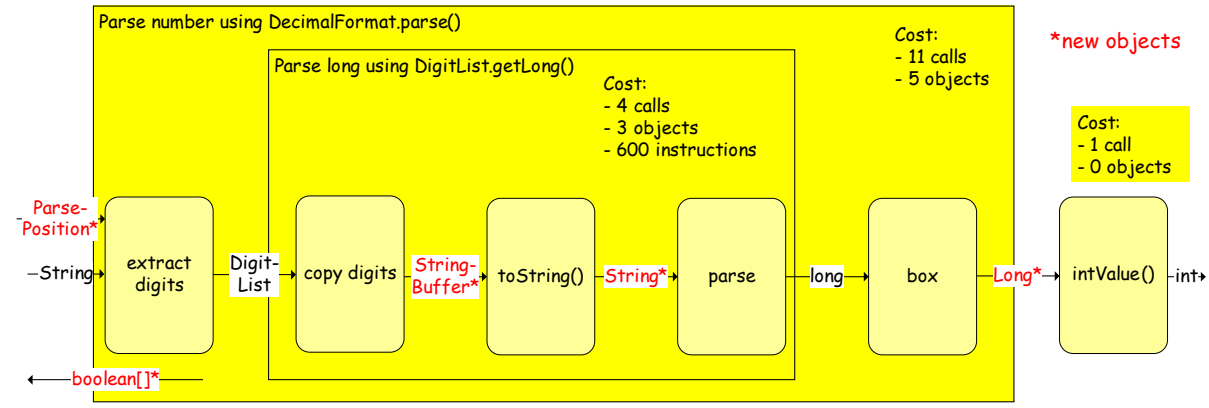

Figure 5. Diary of a Year – Diagram Level 3 (subfield)

4.TOPOLOGICAL METRICS

Measures of the topology of a data flow diagram can give us some clues as to the complexity of an implementation. We can derive various measures from a single level of diagram, such as the total number of transformations and the maximum path length. For example, the first top-level step of converting a date to a business object field in Figure 2 is implemented by a total of ten transformations – a sign that this is not a simple operation.

Other useful measures can be derived by looking at the entire hierarchy of data flow diagrams underlying a given transformation. These can give a sense of how "far afield" an implementation has gone from its high-level interface. Our top-level transformation hides three levels of detail, and takes 58 transformations in total. There are a total of 8 transformations at the first level of depth, 14 at the second, and 36 at the third. This breakdown shows us that much of the activity is delegated to a distant layer.

5.RELATED WORK

Recent work addresses a similar problem to ours: in framework-based applications, the actual coding process is difficult, due to the long chains of frameworks that must be understood [12].

Many works on performance understanding assign measurements to the artifacts of a specific application or framework [2,3,7,8,14,16]. Some have identified that static classes do not capture the dynamic behavior of objects [11].

There is much work on using data flow diagrams, at design time, to capture the flow of information through processes at a conceptual level [6,9]. In contrast, tools and compilers that study program code use a variety of analyses to capture the definitions and uses of program variables [15].

Finally, there is much work on recovering the design of complex applications [4,13].

6.CONCLUSIONS AND DIRECTIONS

That developers make such reuse of frameworks has been a boon for the development of large-scale applications. The flip side seems to be complex and poorly-performing programs. Developers can not make informed design decisions because costs are hidden from them. Moreover, framework designers can not predict the usage of their components. They must either design overly general frameworks, or ones specialized for use cases about which they can only guess.

We believe that elements of forming diagrams and grouping can be automated, for example, by using escape analysis, data flow analysis that combines static and dynamic information, and clustering based on descriptive labels (e.g. ones that identify data structures as records or fields) and application/framework boundaries. Programmers and designers must however remain a critical part of this process. Automation will also enable validation of the approach against a larger set of applications.

7.ACKNOWLEDGMENTS

We wish to thank Tim Klinger, Edith Schonberg, and Kavitha Srinivas for their technical contributions and support of this work.

8.REFERENCES

[1] William P. Alexander, Robert F. Berry, Frank E. Levine, and Robert J. Urquhart, A Unifying Approach To Performance Analysis in the Java Environment, *IBM Systems Journal* Volume 39 Number 1, 2000.

[2] G. Ammons, J. Choi, M. Gupta, and N. Swamy. Finding and Removing Performance Bottlenecks in Large Systems. *ECOOP*, 2004.

[3] E. Arisholm. Dynamic Coupling Measures for Object-Oriented Software. *Symposium on Software Metrics,* 2002.

[4] B. Bellay and H. Gall. An Evaluation of Reverse Engineering Tool Capabilities. *Journal of Software Maintenance:Research and Practice* Volume 10, 1998.

[5] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1, W3C Note 08, 2000.

[6] P. Coad and E. Yourdon. O*bject-Oriented Analysis*, 2nd Edition, Englewood Cliffs, NJ: Prentice-Hall, 1991.

[7] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan. Drive-by Analysis of Running Programs. *Workshop on Software Visualization, ICSE*, 2001.

[8] B. Dufour, K. Driesen, L. J. Hendren, C. Verbrugge. Dynamic Metrics for Java. *OOPSLA* 2003: 149-168.

[9] C. Gane and T. Sarson. *Structured Systems Analysis*. Englewood Cliffs, NJ.: Prentice-Hall, 1979.

[10] IBM Trade Web Application Benchmark http://www.ibm.com/software/webservers/appserv/wpbs_download.html

[11] V. Kuncak, P. Lam, and M. Rinard. Role Analysis. *POPL*, 2002.

[12] D. Mandelin, L. Xiu, R. Bodik, and D. Kimmelman. Mining Jungloids: Helping to Navigate the API Jungle. *PLDI*, 2005.

[13] T. Richner and S. Ducasse. Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. *ICSM*, 2002.

[14] K. Srinivas and H. Srinivasan. Blame Assignment: Summarizing Application Performance from a Components Perspective. FSE 2005.

[15] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 1995

[16] Robert J. Walker, Gail C. Murphy, Jeffrey Steinbok, and Martin P. Robillard. Efficient Mapping of Software System Traces to Architectural Views. In *CASCON*, 2000.