Efficient Mapping of Software System Traces to Architectural Views

Robert J. Walker, Gail C. Murphy, Jeffrey Steinbok, and Martin P. Robillard
Department of Computer Science,
University of British Columbia,
201-2366 Main Mall,
Vancouver, BC V6T 1Z4, Canada

Technical Report TR-00-09

July 7, 2000

Abstract

Information about a software system's execution can help a developer with many tasks, including software testing, performance tuning, and program understanding. In almost all cases, this dynamic information is reported in terms of source-level constructs, such as procedures and methods. For some software engineering tasks, source-level information is not optimal because there is a wide gap between the information presented (i.e., procedures) and the concepts of interest to the software developer (i.e., subsystems). One way to close this gap is to allow developers to investigate the execution information in terms of a higher-level, typically architectural, view. In this paper, we present a straightforward encoding technique for dynamic trace information that makes it tractable and efficient to manipulate a trace from a variety of different architecture-level viewpoints. We also describe how this encoding technique has been used to support the development of two tools: a visualization tool and a path query tool. We present this technique to enable the development of additional tools that manipulate dynamic information at a higher-level than source.

Keywords

Dynamic information, execution information, paths, software integration testing, program understanding, performance analysis, traces, encoding.

1 Introduction

Dynamic information—information about a software system's execution—can help a developer with many different tasks, including software testing [7], performance tuning [4], and program understanding [1]. Since dynamic information is collected either by instrumenting the source code or by modifying the execution environment, the information is fine-grained, reporting on such items as instructions and basic blocks. To help the developer interpret this information, tools typically take this fine-grained information and report it in terms of constructs that the developer is manipulating in the source code, such as procedures.

For some software engineering tasks, source-level information is not optimal because there is a wide gap between the presented information and the concepts of interest to the software developer. For example, when performing some kinds of software integration testing or when reasoning about the impact of some program changes, it may be more natural for a developer to think in terms of subsystems rather than procedures. Manually maintaining the association of source-level information to more abstract concepts such as subsystems is, at best, time-consuming and error-prone. For large systems, manual maintenance of the association may be intractable.

Although tools to help developers manipulate *static* information at a higher level than source have been available for a number of years (e.g., [8]), there has been less work focused on helping de-

velopers interpret and manipulate *dynamic* information from an abstract, typically architectural, view. Those tools that do exist take one of two approaches. The first is to annotate the source code to report the dynamic information in terms of the system's architecture (or other abstract concepts); this approach was taken by Sefika and colleagues in a tool built to report performance information in architectural terms [13]. However, this approach limits both the architectural views that can be used and the means by which the information is collected. The second approach is to allow information to be collected at a fine-grained level and then to be mapped to the architecture-level; we have focused on the latter approach [10, 15].

Specifically, in this second approach, a developer provides a mapping specification that describes how the collected information relates to the abstract level. In the two tools we describe in this paper, the mapping specification consists of an ordered list of pairs of regular expressions and names of architectural components: an entity reported in the dynamic information is considered to be part of the first architectural component whose regular expression it matches. This approach allows a developer to alter the mapping to view the system from different architectural perspectives.

If the dynamic information of interest is a summary of the execution, it is generally reasonable and efficient to map the information after it is collected. For example, if the dynamic information is a summary of the number of times each procedure has been entered, each procedure would only need to be mapped once. However, when the dynamic information is in the form of a trace,[1] it is costly to map each element. In our approach, for instance, we would end up matching each trace element against a potentially large set of regular expressions, resulting in a large number of costly comparisons. Furthermore, if a developer wants to manipulate the dynamic information from more than one architectural view, it may be necessary to duplicate large traces, which may be impractical.

In this paper, we describe a straightforward encoding technique for traces that makes it tractable and efficient to interpret and manipulate a trace, from a variety of architecture-level views. We present this technique to foster discussion and to enable the investigation of the usefulness of manipulating dynamic information at a higher level.

To begin the discussion, we describe the tools we have built upon our encoding scheme to aid the analysis of systems at the architecture-level (Section 2). We then present the process we use to collect traces, our encoding scheme, our approach to mapping encoded traces, and an analysis of the benefits of the encoding scheme (Section 3). We conclude with a short description of why we believe architecture-level traces open new opportunities to develop tools to aid developers in analyzing systems (Section 4).

2 Using Architectural Traces

To investigate whether architectural traces might help developers analyze systems, two tools have been built.

The first tool visualizes dynamic information collected from an object-oriented system. Two small case studies have been conducted on the use of this tool. These studies provided some positive indications that this tool may help developers tune the performance of their system. A brief overview of this tool is provided in Section 2.1; further details are available elsewhere [15].

The second tool supports the extraction of paths between architectural components from trace data. We have not yet performed any studies on the use of this tool beyond applying it to some of the systems we have developed. We describe briefly how this tool might help support integration testing activities.

2.1 Visualization Tool

Our visualization tool allows a developer to analyze the execution of a system off-line. The visualization consists of a temporally-ordered series of pictures, each detailing information about a corresponding point in the execution of the system being analyzed. Rather than displaying raw, low-level events, events are mapped to architectural-level entities as chosen by the developer. Using the visualization tool, a developer can navigate across the trace, either one event at a time or as an animation, seeing how objects mapped to the architectural entities call each other, as well as where objects are allocated and deallocated.[2]

[1] A trace is an ordered sequence of events that occurred during the execution of a system.

[2] The other boxes in the screen shots are histograms that provide a view on the memory usage by an architectural component.

```
category Clustering
    class "ArchClusteringAnalysis"
category ModulesAndSuch
    class "Arch(Procedure|Symbol)"
category SimFunc
    class "ArchSimFunc"
category Rest
    class "Schwanke*"
```

Figure 2: Map Specification for Figure 1

In contrast to many performance analyzers, such as profilers, the visualization of a trace can put information in perspective, showing how and when components of a system interact. Abstracting these interactions to the architecture-level can provide insight into different kinds of performance problems, such as when a subsystem might be using more memory than expected and why that is happening.

Figure 1 shows a screen snapshot of the tool. This snapshot shows a point about halfway through the execution of a sample run of the implementation of a hierarchical agglomerative reverse engineering algorithm [12]. This algorithm attempts to automatically cluster entities, such as procedures in a C program, comprising a software system into subsystems (modules) based on a similarity function. In the visualization, the classes implementing this algorithm are mapped to four architectural entities (the dark boxes): *Clustering*, representing the class performing the clustering analysis; *SimFunc*, representing the class containing methods for computing the similarity function; *ModulesAndSuch*, representing the functions and modules whose similarity was to be compared; and *Rest*, representing all other classes involved in the algorithm. Figure 2 shows the specification created to map low-level events to these architectural entities; each event is compared against the regular expression in the lexical order specified until it matches one, at which point it is mapped to the corresponding architectural entity. This particular visualization was used in a case study that discovered the source of execution problems in the implementation of the reverse engineering algorithm; further details about the case study are available elsewhere [15].

A key property of the visualization tool is its dependence on fast, iterated mapping, or abstraction, to the architectural level. The developer may not have a good idea of what architectural entities to map to initially. Furthermore, even when the developer has a good idea of what architectural entities to use for a given task, that task can change as initial questions are answered, or new questions arise. If the process of specifying the map and performing the abstraction is time-consuming, the usability of the tool suffers markedly. An efficient means of performing the mapping was needed, leading to the development of the encoding technique described in Section 3.

2.2 Path Query Tool

Consider a software developer faced with the task of developing integration test cases for a large system. Hopefully, the developer will have access to various documents describing the system design and implementation that can be used to determine which cases need to be tested. The developer would then proceed to determine inputs and configurations to execute the desired cases. However, how can the developer determine if a particular test case, once executed, does indeed exercise the paths of interest?

To the best of our knowledge, little support is available to help software developers answer this question. Existing coverage tools report information about such items as basic blocks, line, functions, files, directories, and sometimes, libraries and applications.[3] A developer might use this coverage information to gauge which entry points to a subsystem were being exercised, but from this information it would be difficult to determine path information.

Path profile tools can provide more useful information. Although early path profile tools were limited to reporting intra-procedural paths [2], a more recent tool reports inter-procedural path profiles [6]. These inter-procedural path profiles represent a summary of the execution that could help determine path coverage. Summary information as found in these profiles, however, may not always be sufficient. Rather, it might be helpful to understand the relative ordering of paths in an execution and to have, as part of the path, additional events such as object allocations. For instance, in an object-oriented program, it may be important to have one path execute prior to another path to appropriately set the state of a series of objects.

[3] For example, Rational's PureCoverage product can report coverage at line, function, file, and other, levels.

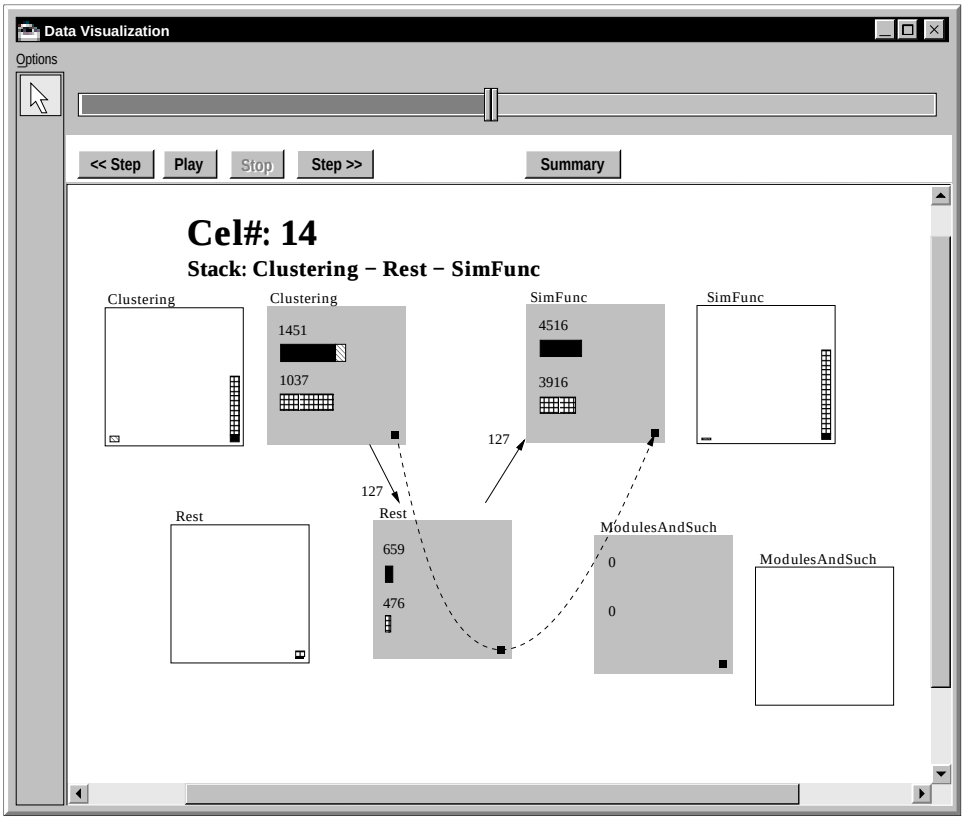

Figure 1: Architecture-level Visualization

To investigate whether detailed path information might help a software developer reason about a set of integration test cases, we have developed a path query tool that operates on trace data and that supports queries at the architecture-level. Given a trace and a mapping specification (similar to that shown in Figure 2) describing how source-level components relate to architectural components, the tool will extract all paths starting in one named architectural component and ending with an entry to a second named architectural component. The paths extracted contain both call information and object allocation and deallocation information. Sub-paths are also reported.

To try out this tool, we applied it to analyze some test cases for the Jex static analysis tool [11]. Jex analyzes the flow of exceptions in Java™ programs and consists of over 100 classes. Six architectural entities comprise Jex: a Controller, a Parser, a Type system, an AST, a Loader for reading intermediate files, and a utility subsystem. In our trial use of the path query tool, we were interested in the paths exercised between the AST and the Loader component by three test cases. We used the path query tool to extract the paths between these two architectural entities: 534 paths were found. We analyzed these paths to determine if they were indeed the paths intended to be exercised. Our analysis showed that one of the test cases exercises a greater variety of paths than the other two test cases. Specifically, one case ensures that the Loader component is called in three different situations: while processing method invocation expressions, while processing `throw` expressions, and while processing other Java expressions. The other test cases focus only on the latter situation. This information may be useful to help assess and select test cases. Furthermore, one might care about invoking the Loader from a method invocation prior to a `throw` expression; the path query tool can help you determine if a test case meets this criterion.

The ability of our tool to understand the mapping between the source and architectural components

makes it easy for a software developer to extract the paths of interest. Instead of having to translate the questions of interest for the software integration testing task, a developer can express the questions directly in terms of the components being integrated. Once relevant paths have been extracted using this approach, a variety of further analyses can be performed. For instance, the paths could be viewed using a browser similar to the Hot Path Browser by Ball and colleagues [3], or could be analyzed using concept analysis as also described by Ball [1].[4]

As with our visualization tool, the developer may need to iterate the mapping specification to refine it to answer the test case question of interest. For instance, as the developer learns about the different possible courses of execution, the developer may wish to refine subsystem boundaries. As before, then, fast, iterated abstraction is a must here, hence the need for the encoding technique described in Section 3.

3 Mapping Traces

Both of the tools described rely on trace information collected from a program's execution. Program trace information has been used for many years and a number of techniques have been developed for collecting and storing it [5]. These efforts focus on the efficient collection and representation of detailed information about the program, such as the instructions executed and the data locations referenced. These detailed traces help support the design of memory systems and help guide the behaviour of parallelizing compilers, amongst other uses.

In comparison, the traces we use support software engineering activities. We can support these activities using less detailed traces. In the object-oriented systems we have been studying, our traces consist of information about message sends, object allocations, and object deallocations. Although this information is already at a higher level than program instructions, we believe software developers dealing with large systems can benefit from further abstraction of the information.

Trace information is collected in one of three ways: by instrumenting source files, by instrumenting object files, or by altering the execution environment.[5] The framework we have developed encodes objects, representing events of interest that occur during execution, in the format described below.

In this section, we describe our trace representation. First, we describe the events we are recording and how we encode these events in the trace representation. Next, we describe how the encoding facilitates the abstraction and summarization of the events. Finally, we describe why this encoding scheme is of benefit.

3.1 Events

The traces we are collecting describe the execution of an object-oriented system. Traces compose the following types of events:

- class method entry and exit events,
- instance method entry and exit events,
- object allocation and deallocation events, and
- thread start and stop events.

Each event carries particular information relevant to the system event it represents. Class method entry and exit events record the name of the class and the name of the method that was entered or exited (class and method identifiers). Instance method entry and exit events record an additional identifier representing the object on which the method was called. Object allocation and deallocation events record a class identifier and an object identifier. All of these event types also record the name of the thread executing the event (thread identifier). Finally, the thread start and stop events record a thread identifier.

3.2 Encoding Events

As with any encoding scheme, the key lies in determining the patterns that can be used to encode the information of interest. Since our goal was to abstract each event, we needed to determine how to support the abstraction operation efficiently. The

[4] Our trace information does include timestamps so the durations of paths can be determined.

[5] Our current set of tools works on Java programs. We are using AspectJ™ from Xerox PARC (`http://www.aspectj.org/`) to instrument Java source, and the Jikes Bytecode Toolkit from IBM Research (`http://www.alphaworks.ibm.com/`) to instrument bytecode. We have also created a translator to convert IBM Research's Jinsight traces (`http://www.alphaworks.ibm.com/`), which are produced by the Jinsight VM, to our trace format.

abstraction operation consists of testing an event to see whether it meets some set of properties associated with the description of an abstract item. For instance, in the tools described above, the association between an event and an abstract item consists of a set of regular expressions; an event is associated with the abstract item if it matches one of the regular expressions.

Our encoding scheme meets this goal by categorizing events and encoding the categories in the trace. With this encoding scheme, we record traces in two streams: an encoding stream, and an event stream. The encoding stream consists of a sequence of records, each containing information about a given category; these categories are termed *primitive* because they cannot be subdivided. The event stream consists of a sequence of records, each of which contains an index to a primitive category within the encoding stream, plus some additional information that depends on the type of event involved.

A primitive category consists of a unique combination of class identifier, method identifier, thread identifier, and event type. Primitive categories do not include object identifier information because, in general, the number of events associated with a given object will be small, and therefore, the number of primitive categories with which we would have to deal would increase dramatically. Events that contain object identifiers record them within the event stream.

Figure 3 demonstrates this encoding scheme. The event stream starts with a `ClassMethodEntryEvent`. The details about this event, such as the class and method that were entered and the thread in which the method was executed, are recorded on the encoding stream. The record on the event stream includes the ordinal number of the full information on the event encoding stream. When the second `ClassMethodEntryEvent` occurs, it is a call to the same class and method in the same thread as the first event; therefore, we encode it in the event stream as being the same category, and nothing new is written to the encoding stream. The `InstanceMethodEntryEvent` that occurs later in the event stream is encoded similarly to the first event. Unlike the `ClassMethodEntryEvent`, the record on the event stream for the entry of an instance method includes information about the object on which the method was invoked.

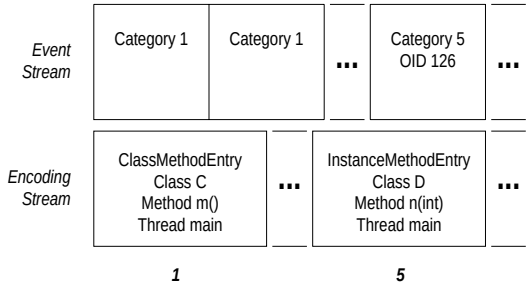

Figure 3: Encoding Scheme

3.3 Abstracting Events

Interpreting a trace at an abstract level requires applying an abstraction operation to each event in the trace. Encoding the event stream facilitates this interpretation.

Instead of having to apply the abstraction operation against each event, the abstraction operation need only be applied against each record in the encoding stream, i.e., each primitive category. The architecture-level entities to which they are to be mapped are termed *abstract categories*.

For each tool, the developer using it specifies a mapping from a set of primitive categories to an abstract category through a partial, ordered specification of matching criteria. For example, in Figure 2, the developer specified that any events referring to the class `ArchClusteringAnalysis` should be mapped to the *Clustering* abstract category. This means that each encoding stream record has its class identifier (if any) compared against this matching criterion. If it matches, the event is placed in the *Clustering* abstract category; if not, the event is then compared against the next mapping criterion. If the event matches none of the criteria, it is not mapped, and is not used further.

The abstraction operation produces an array of values: the primitive category number serves as an index into the array, which stores the abstract category to which each primitive category is to be mapped. In the example in Figure 2, we might have had hundreds of primitive categories, but only four abstract categories, so our array would have been (identically) hundreds of elements in size, but each element would reference an abstract category as a number from 1 to 4, or 0 if it was not mapped at all. The larger event stream can then be traversed, and each individual event, which refers to its primitive category, can be mapped to the appropriate abstract

category via an $O(1)$ lookup in this array.

3.4 Summarizing Events

Software developers can also benefit from the summarization of events: summarization abstracts the events over time. For example, as described earlier in this paper, path profile tools summarize the paths taken during an execution [2, 6].

Summarization and abstraction of events are orthogonal techniques. Although each is useful on its own, their combination can provide further software analysis support. For example, to help software developers understand a trace, our visualization tool summarizes, throughout the trace, the number of objects allocated and deallocated that belong to each abstract category.

Our encoding scheme facilitates the combination of these techniques by allowing the most costly part of summarization to occur once, prior to abstraction. Summarization is performed with respect to individual primitive categories and recorded. Later, these recorded summaries can be abstracted by applying the abstraction operation to the primitive categories in the summary, and then, for each abstract category, aggregating the summarizations of the primitive categories that map to it.[6] Since many events may map to a primitive category, this two-step process allows the abstraction to be altered much more cheaply than re-summarizing in a single step would.

For example, if we found that 32 instances of `String` and 14 instances of `StringBuffer` had been allocated during a trace and the architectural view called for all `String` and `StringBuffer` events to be grouped together into the `StringOp` abstract category, we would simply add the two counts to find that 46 objects were allocated in the trace that mapped to `StringOp`.

Without the notion of indivisible, primitive categories, as found in our encoding scheme, each event could be mapped arbitrarily to an abstract category. This would prevent any partial summarization from being performed prior to abstraction. Since summarization over a trace requires processing of the entire trace, if the architectural view of the system is to be changed frequently, as it is in our model, summarization can be a prohibitively expensive operation.

[6] This aggregative scheme assumes that the total summarization in question is describable solely as a function of abstract category.

3.5 Savings

The encoding strategy is only an advantage if two conditions are met: (1) primitive category information tends to be repeated in the trace, and (2) the abstraction operation is costly to perform.

The first condition is important since we will only gain an advantage if the encoding stream is smaller than the event stream. This condition will typically hold: the number of events produced when running a system is large compared with the number of classes and methods in a system, upon which the encoding scheme is based. The total number of encodings possible for a given system is a small multiplier of the product of the number of classes and the number of methods and the number of threads. As one example, for the Jex tool described in Section 2, Jex produced a trace composing 5×10^5 events as it analyzed one simple Java class. Encoding this trace results in only 725 primitive categories.

The second condition matters because all events in the trace still require processing. When the abstraction operation is cheap to perform, it may as well be applied as the events are traversed. However, when the abstraction operation is expensive, it is an advantage to apply it only to the much smaller number of encodings. At first glance, our regular expression-based comparison may appear cheap since an individual regular expression comparison is not necessarily costly. Although we do not yet have much experience with applying the regular expression-based operation against trace data, when applying it to static data collected from the source code of Microsoft Excel to support an experimental reengineering task, the number of comparisons grew to be large, over 1000 in total [9]. Obviously in such a case, comparing against the primitive categories rather than the events results in a more efficient tool. This savings also provides an opportunity to try out more expensive abstraction operations, such as operations involving some inference.

To clarify the savings of the encoding scheme, consider that the cost of abstracting a trace is on the order of $\sum e_i p_i + a_i$ where e_i is the number of events belonging to primitive category i, p_i is the cost of identifying that a given event belongs to primitive category i, and a_i is the cost of applying the abstraction operation to primitive category i. Without the encoding scheme, we can still consider the set of events that would have belonged

to primitive category i, for the sake of our analysis. In the absence of the encoding, the abstraction operation has to be performed on each event instead of once for the entire primitive category for a total cost of $\sum a_i e_i$.[7] The savings in using the encoding scheme is on the order of $\sum (a_i - p_i)e_i - a_i$. The encoding scheme will thus be an advantage when the conditions above are met.

4 Summary

Can the abstraction and summarization of trace information enable new software analysis approaches? Can it enhance existing approaches? Can it help software developers perform software engineering tasks more effectively?

There are no definitive answers to these questions—yet. To answer these questions, it is necessary to have the base technology to abstract and summarize traces efficiently. This technology allows tools to be built that can be applied to realistic systems and realistic scenarios.

This paper has presented an encoding scheme that provides this base technology. Traces may be abstracted to different architectural views. Trace information may also be intermittently summarized and then abstracted.

Although, to date, we have only limited experience with applying this technology, we believe it holds promise for increasing the usefulness of dynamic information in software engineering tools and techniques. As an example, in addition to the visualization and path query tools we have built, the approach may enable the determination of architectural dependences between pieces of existing systems [14]. This information could enable a new way to verify that a system adheres to its architectural goals.

Acknowledgments

This research was funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and in part by the Consortium for Software Engineering Research (CSER) in cooperation with Object Technology International, Inc.

"AspectJ" is a trademark of Xerox Corporation. "Java" is a trademark of Sun Microsystems.

[7] Abstracting an event directly to an abstract category will cost the same as abstracting an event from a primitive category. The actual abstraction process is a regular expression matching that could be performed on either events or primitive category records identically.

About the Authors

Robert J. Walker is a Ph.D. candidate in the Department of Computer Science at the University of British Columbia. His thesis work concerns the use of dynamic contextual information for software evolution and reuse. He may be contacted at walker@cs.ubc.ca.

Gail C. Murphy is an assistant professor in the Department of Computer Science at the University of British Columbia. Her research interests are in software evolution, software design, and source code analysis. She may be contacted at murphy@cs.ubc.ca.

Jeffrey Steinbok is a recent graduate from the University of British Columbia. He currently works at Microsoft. He may be contacted at steinbok@cs.ubc.ca.

Martin P. Robillard is a Ph.D. student in the Department of Computer Science at the University of British Columbia. His research interests include program understanding, evolution, and modularization. He may be contacted at mrobilla@cs.ubc.ca.

References

[1] Thomas Ball. The concept of dynamic analysis. In Oscar Nierstrasz and Michel Lemoine, editors. *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, Toulouse, France, 6–10 September 1999, pp. 216–234. 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[2] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 46–57, Paris, France, 2–4 December 1996.

[3] Thomas Ball, James R. Larus, and Genevieve Rosay. Analyzing path profiles with the Hot Path Browser. In *Workshop on Profile and Feedback-Directed Compilation*, Paris, France, 13 October 1998. `http://www-cse.ucsd.edu/users/calder/fdo/`

archive/fdo1/papers/pfdc-ball.ps.Z.

[4] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 120–126, Boston, Massachusetts, USA, 23–25 June 1982.

[5] James R. Larus. Efficient program tracing. *Computer*, 26(5):52–61, 1993.

[6] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pp. 259–269, Atlanta, Georgia, USA, 1–4 May 1999.

[7] Edward F. Miller, Jr. Program testing: Art meets theory. *Computer*, 10(7):42–51, July 1977.

[8] Hausi A. Müller and Karl Klashinsky. Rigi—A system for programming in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pp. 80–87, Singapore, 11–15 April 1988.

[9] Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *Computer*, 30(8):29–36, August 1997.

[10] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between design and implementation. To appear in *IEEE Transactions on Software Engineering*, 2000.

[11] Martin P. Robillard and Gail C. Murphy. Analyzing exception flow in Java™ programs. In Oscar Nierstrasz and Michel Lemoine, editors. *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, Toulouse, France, 6–10 September 1999, pp. 322–337. 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[12] Robert W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pp. 83–92, Austin, Texas, USA, 13–17 May 1991.

[13] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th International Conference on Software Engineering*, pp. 387–396, Berlin, Germany, 25–29 March 1996.

[14] Judith A. Stafford, Debra J. Richardson, and Alexander L. Wolf. Architecture-level dependence analysis for software systems. In *International Workshop on the Role of Software Architecture in Testing and Analysis*, Marsala, Sicily, Italy, 30 June–3 July 1998. http://www.ics.uci.edu/~djr/rosatea/papers/stafford.pdf.

[15] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 271–283, Vancouver, British Columbia, Canada, 18–22 October 1998.