Dynamic Analysis for Locating Product Features in Ada Code

Laura J. White
Department of Computer Science
University of West Florida
11000 University Parkway
Pensacola, FL 32514
(850) 474-3017
lwhite@uwf.edu

Norman Wilde
Department of Computer Science
University of West Florida
11000 University Parkway
Pensacola, FL 32514
(850) 474-2548
nwilde@uwf.edu

1. ABSTRACT

In this paper, we describe a Software Reconnaissance method for locating product features in code. The University of West Florida has provided a public domain Reconnaissance tool for C and C++. Characteristics of the Ada programming language present new interesting issues related to the development of a similar tool for Ada Reconnaissance, particularly its use in embedded systems, multi-tasking systems, and systems with real-time constraints. We describe our on-going efforts to develop a public domain Ada Reconnaissance tool.

1.1 Keywords

Software Reconnaissance, instrumentation, dynamic analysis, locating features in code, Ada tracing architecture, trace monitoring, performance trials.

2. INTRODUCTION

One of the most difficult tasks facing any Software Engineer is that of updating an unfamiliar program. Despite years of development of Software Engineering techniques, the job still somehow comes down to long hours of paging through code in an editor or debugger.

Practicing Software Engineers tell us that reliably locating *product features* in unfamiliar programs can be one of their biggest headaches. Our research in feature location was motivated several years ago by a visit to a site that maintained the software of a telephone PBX switch. To make changes correctly, maintainers needed to find all the code fragments involved in end-user features such as "call forwarding" or "call waiting". In a large and frequently modified system such as this one, the code for a feature is often not contiguous or located in obvious places.

In response to this need, we developed a simple but elegant dynamic analysis method called *Software Reconnaissance* to aid in locating such product features. The method has been tried out and refined with support from the Software Engineering Research Center (SERC)[1]. Case studies at several sites indicate that the method seems to be useful as described in an earlier paper by Wilde[18]. For C software, students at the University of West Florida have developed a public domain Reconnaissance tool called Recon2 [10] and Telcordia, a SERC affiliate, has included a Reconnaissance facility in their χSUDS testing toolkit for C and C++ described by Telecordia in [14].

Several SERC industrial affiliates also expressed an interest in applying Reconnaissance to Ada code, and particularly to Ada embedded systems. However such code has characteristics that can make dynamic analysis difficult. This paper describes the progress of our current efforts to address the issues involved in extending Software Reconnaissance to Ada and to develop a public domain Ada Software Reconnaissance tool.

[1] SERC is an NSF Industry/University Cooperative Research Center, with sites at Purdue University, the University of Florida, the University of Oregon, and the University of West Virginia. SERC, and this research, has been supported by NSF grant EEC-9418762 and others, and by SERC industrial affiliates.

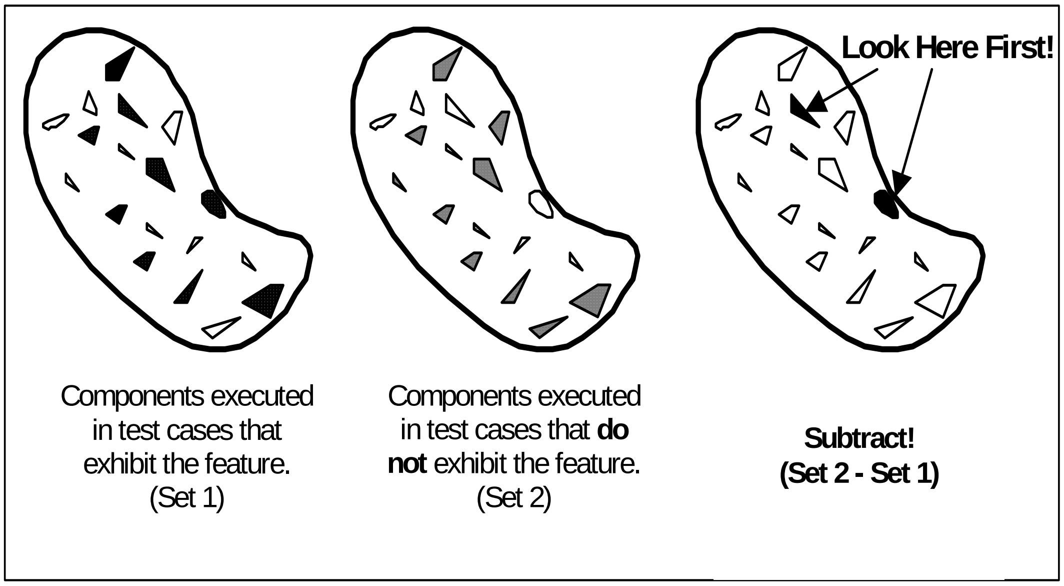
Look Here First!
Components executed in test cases that exhibit the feature. (Set 1)
Components executed in test cases that **do not** exhibit the feature. (Set 2)
Subtract! (Set 2 - Set 1)

Figure 1.

Locating "Marker" Components for a Feature

3. THE SOFTWARE RECONNAISSANCE METHOD

Software Reconnaissance uses test cases as probes to locate code for a particular product feature. The program is first instrumented in much the same way as programs are instrumented to determine test coverage. Then it is run with a few test cases that exhibit the desired feature and with a few others that do not. The executions of the instrumented code produces trace files showing which code components (usually branches or basic blocks) were used in each test.

The Software Reconnaissance method is described formally by Wilde in [17]. Any one of several different heuristics described there may then be used to locate the components for a particular feature. The simplest, and often the best, is simply to take the set of *marker* components for the feature, defined as those components that are executed in at least one test case that exhibits the feature and not executed in any test case without the feature (Figure 1).

The techniques in the literature that most closely resemble Software Reconnaissance are methods for debugging. A common witticism in Software Engineering is that a program fault may be viewed of as simply an undesired "feature". In desperation, experienced debuggers sometimes resort to a technique called "dump and diff" that involves inserting print statements and running a file difference utility to compare different runs. The first published suggestion that trace information could be used systematically for fault localization seems to have been by Collofello and Cousin [2] who suggested tracing "decision-to-decision paths" and comparing runs that exhibit a failure with runs that do not.

4. EXPERIENCE FROM C TRIALS OF SOFTWARE RECONNAISSANCE

The Software Reconnaissance method of locating features has been tried in a series of case studies, most of them involving C code. The first trials used University or public domain code, as a "sanity check" to see if the method made sense. One of these trials was a protocol study in which experienced C programmers were asked to "think aloud" as they used the method to solve a problem. Such trials are useful in evaluating the usability of a technique by its prospective users, and is described by Wilde in [17].

Additional trials used small software systems (from about 10 KLOC to 50 KLOC, raw line counts) from SERC affiliates and other companies as described in [18]. In these trials a Software Engineer familiar with the system was asked to locate features that he had recently needed to study, or that could be important for a new maintainer. These trials gave us increased confidence in the value of the method, as well as insight into the kinds of test cases that are most effective.

The results of these trials have generally been favorable. Software Reconnaissance does not locate all the code a maintainer needs to study to understand a feature, but it usually finds very good starting points for code exploration in an unfamiliar system. It focuses attention on a relatively small part of the code and can often provide insights that surprise programmers, even if they thought they already understood the program. "I didn't know it was doing that!" is a comment we have heard several times in our studies.

5. SOFTWARE RECONNAISSANCE FOR ADA

These experiences with C would seem to show that an Ada Software Reconnaissance tool would be desirable. However Ada systems have characteristics that make the production of traces somewhat problematic:

- Ada systems are often embedded, which makes it more difficult to produce and store trace files.
- Ada systems often use multi-tasking, so naive methods of tracing, such as direct writes to the file system, may fail due to task contention.
- Ada systems often have real-time constraints, which may limit the amount of tracing which can be performed.

An Ada tool for Software Reconnaissance will thus require careful design. Our approach to developing such a tool has included a survey to identify current instrumentation practices for embedded systems, design of an architecture to accommodate multi-tasking as well as the results from the survey, and timing trials to estimate the performance impact of instrumentation.

6. INSTRUMENTATION PRACTICES FOR EMBEDDED SYSTEMS

There is considerable literature on the problems of tracing or monitoring embedded, real-time and distributed systems. An extensive survey of the debugging problems is given by McDowell and Helmbold in [19]. Schutz also gives an extensive review of the problems, emphasizes the problem of observability, and comments on the difficulties of using conventional debuggers [12]. He also describes several monitoring systems.

Many other authors also discuss the theory of monitoring methods and/or describe specific monitoring or debugging systems, for example [8, 15, 11, 13]. Yan, Sarukkai and Mehra describe some experience with a very extensive set of monitoring and visualization tools from NASA's Ames Research Center [20]. Hollingsworth, Miller, Goncalves, Naim, Xu and Zheng describe a technique they call "dynamic program instrumentation" in which the running program is modified periodically to collect information about its execution in [6]. Waheed, Rover and Hollingsworth give a detailed analysis of some of the design alternatives in monitoring distributed systems, such as the trade-off between sending event data immediately or batching it for efficiency in [16]. Heath and Etheridge describe methods for graphically displaying the results of monitoring parallel systems, a topic that is very important for effective program comprehension in [5].

Our group took a different tack and surveyed practicing Software Engineers, mostly from SERC affiliate companies, to see what instrumentation methods were actually being used. The survey covered 10 embedded systems ranging from 2 KLOC to 3 MLOC in size, and from mid 70's to late 90's in development epoch. Only two of the systems were written in Ada, but they were all embedded and had real-time constraints. A more complete report of the survey is given in [19].

In general, we found that instrumentation was often ad-hoc, and was only added after severe problems were encountered in integration or testing. Few engineers were aware of or had available to them instrumentation systems of the sophistication described in the studies mentioned previously.

The survey clearly showed that extracting trace data from an embedded system is difficult; there can be no general solution because the hardware differs so greatly from system to system. Software engineers exercised great ingenuity to extract trace information. Methods described to us used everything from light emitting diodes to logic analyzers to conventional flat files depending on the available output devices.

We also found that our interviewees wanted to go considerably beyond the needs of Software Reconnaissance. Reconnaissance only requires a trace that shows if a particular component (e.g. subprogram or basic block) was executed or not. However for understanding embedded systems it is often essential to know the number of times the component was executed, the actual sequence of execution, or possibly even the time at which execution occurred.

Based on these interviews, we defined three primary circumstances in which a Software Engineer could make use of Ada Software Reconnaissance:

- Testing in a development environment where timing constraints are not stringent. Reconnaissance could help acquire an understanding of the features of a legacy software system.
- Testing using a monitor connected to the actual target system. Traces would be sent to the monitor by a variety of methods (socket, communications port, logic analyzer, etc.) depending on the target system hardware available. Reconnaissance could be used to analyze and evaluate run-time behavior of the target system.
- Remote monitoring of a deployed system for troubleshooting purposes, making use of instrumentation previously installed. Reconnaissance could be used to highlight changes in behavior as the system executes as described in [7].

There are thus three decisions to be made for each tracing session:

- What events are instrumented (e. g. package initializations, subprogram entries/returns, basic block executions, task activation and rendezvous?)
- What information is captured at each event (e.g.just whether the event occurred, a count of occurrences, the sequences of occurrences, or the time of the occurrances)
- How trace monitoring is performed (e.g. via a socket, a logic analyzer, a communications port, etc.)

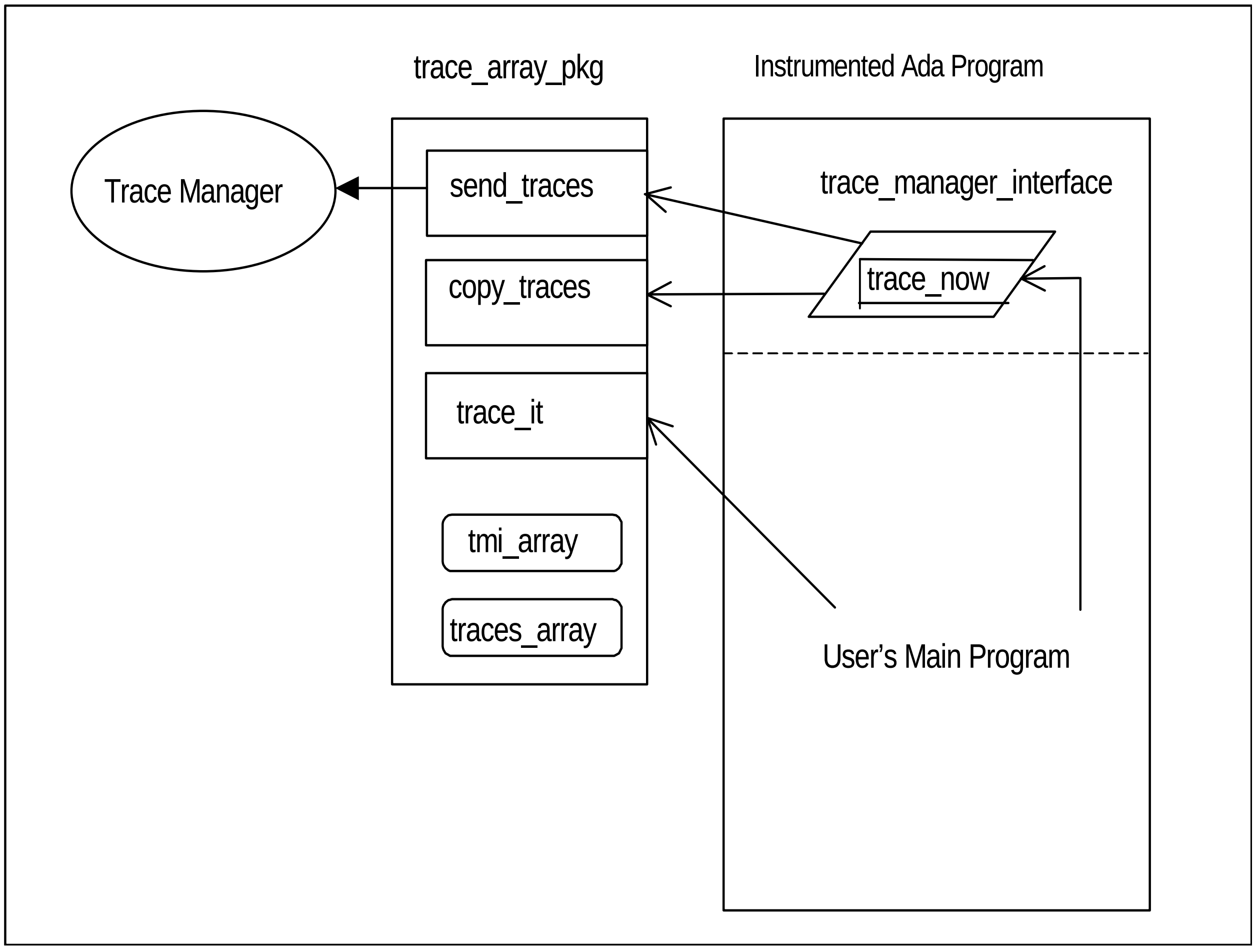

Figure 2.
General Architecture for Ada Tracing

7. THE ADA TRACING ARCHITECTURE

It is clear that no one solution will meet all these needs. Instead, we have defined a set of interchangeable components that can be used to instrument an Ada program. The general architecture is shown in Figure 2.

The instrumented program actually contains the `trace_manager_interface` task embedded in the declarative part of the main program unit. Trace data is stored within the `trace_array_pkg` which is made visible using a `with` context clause. Additional instrumentation which calls the `trace_it` procedure is used to record each trace event in the `traces_array`, which is declared using the `Atomic_Components` pragma to prevent contention.

The `trace_manager_interface` task runs concurrently with the users program units. At the `trace_now` entry call, `copy_traces` copies the `traces_array` to the `tmi_array` during the rendezvous, after which `send_traces` and the users tasks continue concurrently.

The `send_traces` procedure would have to be written specially for each environment, since the mechanism for trace monitoring is hardware-dependent. The trace manager may be as simple as a program to write trace data to a file, or it could interface to a more sophisticated analysis or display tool.

8. PERFORMANCE TRIALS FOR ADA TRACING

All methods of producing traces may be to some extent *intrusive*, that is, they may change the behavior of the system being studied. For real-time software, intrusion is a key issue, since the behavior of the instrumented system may not be the same as that of the original, and the conclusions of the analysis may thus become invalid.

Intrusion is less important in Software Reconnaissance than in other kinds of analysis since the main goal is simply to locate code that implements a feature; the analysis of the feature is performed by a human studying the identified code. Unless system behavior is so radically changed by the instrumentation that the feature is not performed, we should still locate it. If necessary, instrumentation may be disabled for some of the tightest loops.

However it would certainly be useful to have a rough idea of the performance impact of the different kinds of trace instrumentation, at least to give Software Engineers a reference point in deciding what strategies they might try. Accordingly we carried out a performance study using several small, hand-instrumented Ada programs that are publicly available.

The problems with basing decisions on any kind of time benchmarking are well known. For example [1] describes the difficulties in developing a set of benchmarks for different features of Ada. The programs and the environment used for the benchmark may not be representative of your system so "your mileage may vary".

Table 1 lists the ten programs used in the performance trials. These were small Ada samples found on the internet generally intended to illustrate some programming technique or feature of the language. Since execution times were very short, in most cases a loop was added to repeat the body of the code thus giving a runtime of 15 to 50 seconds. Runtime was computed as the difference between system time at the start and at the end of execution. Since an isolated machine was not available, the trials were carried out using somewhat obsolete Sun Sparc computers in a student lab, but at late night hours when little other use was expected. To wash out any effects of differing system overhead, each trial was repeated ten times and the runtimes were averaged. In most cases the standard deviation of the runtimes was quite low.

The programs were compiled using GNAT with full optimization, since that would most likely be used by a software engineer concerned about performance.

The architecture of Figure 2 was implemented using two separate computers, one running the trial program and the `trace_array_pkg` package, and a second acting as a trace manager to collect the generated traces. The `trace_now` entry point was called just once at the end of each run to trigger `copy_traces` and `send_traces`. The `send_traces` procedure used a TCP/IP socket to transmit the trace to the second machine.

To explore the effect of instrumentation design decisions the following were varied:

1. Level of instrumentation events: Instrumentation was placed either just on subprogram entry and return points or on every basic block. We would expect basic block instrumentation to generate many more calls to `trace_it` since most subprograms consist of more than one basic block and may contain tight loops.

2. Information captured at each event: The choices were Boolean (BOOL) instrumentation which merely captures if the event has occurred at least once, Count (CNT) instrumentation, which counts how many

Error! No index entries found.	**Description**	**Source**
`ACESSALL.ADB`	Demonstrates access types	[CORO]
`BTREE.ADB`	Binary tree demonstration	[CORO]
`CHAR_LOO.ADB`	Loop printing characters	[EMBR]
`FUNCRECR.ADB`	Recursive factorial calculation	[CORO]
`GODYNSTR.ADB`	Dynamic string package	[CORO]
`INPUT_TE.ADB`	Input of different data types	[EMBR]
`LINKLIST.ADB`	Linked list of characters	[CORO]
`MULTOUT.ADB`	File output	[CORO]
`OPEROVER.ADB`	Operator overloading	[CORO]
`RANDOM_T.ADB`	Test random number generator	[EMBR]

Table 1
Programs Used in the Performance Trials

times the event has occurred, Sequential (SEQ) instrumentation which keeps track of the exact sequence of events, so that each event may be stored many times, and Timed (TIME) instrumentation, in which the system clock is read at each event and actual event times are stored. For Sequential and Timed instrumentation, each event was stored at most 1000 times; we doubt that more repetitions would be useful for most kinds of program comprehension analysis. In any case, we would expect that Timed and Sequential would require more time than Boolean or Count, since the processing of each event is more complex.

3. Global or packaged trace array: For Boolean and Count instrumentation we also tried making the trace array global (GLOB) instead of keeping it in a package (PKG). A global array eliminates the overhead of a procedure call at each trace event, and it was thought that this could reduce the performance impact of tracing.

The results of the performance trials are summarized in Table 2. The indicator used is the percentage of slowdown of each program defined as:

$$\frac{100 * (InstrumentedRuntime - UninstrumentedRuntime)}{UninstrumentedRuntime}$$

Instrumentation certainly has an impact on performance, varying from program to program but typically ranging Since this program does quite a lot of file output, we hypothesize that the runtime is dominated by the file operations so instrumentation has little effect.

Perhaps the most surprising result is that the slowdown is almost constant for each program; the type of instrumentation has relatively little effect. We would hypothesize that the time required to create and use the socket connection to the trace manager is the most substantial contributor to the slowdown, so that the decision on the actual type of instrumentation is relatively unimportant. Possibly the key decision in instrumentation design, concerns the way that trace data will be extracted from the running system. Of course, our results must be considered indicative only. Performance impacts may be quite different in other environments and with larger programs.

9. CONCLUSIONS

We have described the Software Reconnaissance method for locating product features in software systems as well as our experience so far with this method as applied to C programs. We believe that Reconnaissance can also be very useful to developers and maintainers of Ada systems.

To create an Ada Reconnaissance tool we have surveyed practicing Software Engineers to see how they use instrumentation with embedded software today. Based on these results, and on our earlier experience with C, we have attempted to define an Ada tracing architecture that will be flexible enough to be usable in the kinds of situations that arise in practice.

We think we are well on the way to providing a public domain Ada Reconnaissance tool that will be made available to the Ada community.

	Basic Block Events						Subroutine Entry and Return Events					
	BOOL		CNT		SEQ	TIME	BOOL		CNT		SEQ	TIME
Program	GLOB	PKG	GLOB	PKG	PKG	PKG	GLOB	PKG	GLOB	PKG	PKG	PKG
ACCESSALL.ADB	39%	38%	37%	37%	37%	36%	39%	37%	36%	37%	36%	36%
BTREE.ADB	27%	31%	28%	32%	34%	31%	26%	31%	27%	31%	35%	35%
CHAR_LOO.ADB	13%	23%	22%	20%	24%	23%	11%	20%	18%	20%	21%	20%
FUNCRECR.ADB	37%	41%	38%	42%	41%	41%	33%	41%	38%	41%	42%	40%
GODYNSTR.ADB	16%	22%	21%	21%	32%	31%	14%	18%	18%	19%	25%	23%
INPUT_TE.ADB	56%	56%	55%	56%	56%	55%	55%	55%	55%	56%	57%	56%
LINKLIST.ADB	31%	34%	28%	34%	36%	33%	27%	31%	27%	31%	31%	32%
MULTOUT.ADB	0%	17%	14%	9%	0%	10%	0%	14%	13%	18%	15%	21%
OPEROVER.ADB	45%	40%	43%	42%	44%	45%	44%	40%	42%	44%	44%	44%
RANDOM_T.ADB	25%	28%	23%	27%	28%	28%	25%	25%	22%	28%	26%	26%

Table 2
Percentage Slowdown for Different Types of Instrumentation

10. ACKNOWLEDGEMENTS

We would like to thank very much all the students who have participated in different facets of this project over the last few years and particularly Gus Lorberg and Andre Wacaster, for their assistance with the architecture and Barry Coker and Carlos Trani for their work on the performance trials. We would also like to thank Dean Knudson for his assistance in the survey of industrial instrumentation practices.

11. REFERENCES

[1] Clapp, R.; Duchesneau, L.; Volz, R.; Mudge, T., and Schultze, T., "Toward Real-Time Performance Benchmarks For Ada", *Communications of the ACM*, Volume 29, Number 8, August 1986, pp. 760-778.

[2] Collofello, James, and Cousin, Larry, "Towards automatic software fault location through decision-to-decision path analysis", *Proceedings National Computer Conference 1987*, pp. 539-544.

[3] Coronado Enterprises, ADA 95 Tutorial, http://www.swcp.com/~dodrill/a95doc/a95list.htm.

[4] Embry Riddle Computer Science Department, http://erau.db.erau.edu/~korn/

[5] M. Heath and J. Etheridge, "Visualizing the Performance of Parallel Programs", *IEEE Software*, Vol. 8, No. 5, (September 1991), pp. 29 - 39.

[6] J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, L. Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation", *PACT'97 - 1997 International Conference on Parallel Architectures and Compilation Techniques*, November 11-15, 1997, San Francisco

[7] K. Lukoit, N. Wilde, S. Stowell, T. Hennessey, "TraceGraph: Immediate Visual Location of Software Features", *Proc. International Conference on Software Maintenance - ICSM 2000*, IEEE Computer Society, Los Alamitos, CA, October 2000, pp. 33 - 39.

[8] D. Marinescu, J. Lumpp, T. Casavant, and H. J. Siegel, "Models for Monitoring and Debugging Tools for Parallel and Distributed Software", *Journal of Parallel and Distributed Computing*, Vol. 9, No. 2, (June 1990), pp. 171 - 183.

[9] C. McDowell and D. Helmbold, "Debugging Concurrent Programs", *ACM Computing Surveys*, Vol. 21, No. 4, (December 1989), pp. 593 - 622.

[10] "Recon Tool for C Programmers", http://www.cs.uwf.edu/~recon/.

[11] U. Schmid, "Monitoring Distributed Real-Time Systems", *Real-Time Systems*, Vol. 7, No. 1, (July 1994), pp. 33 - 56.

[12] W. Shutz, "Fundamental Issues in Testing Distributed Real-Time Systems", *Real-Time Systems*, Vol. 7, No. 2, (September 1994), pp. 129 - 157.

[13] R. S. Side and G. C. Shoja, "A Debugger for Distributed Programs", *Software - Practice and Experience*, Vol. 24, No. 5, (May 1994), pp. 507 - 525.

[14] "Telcordia Software Visualization and Analysis Toolsuite", http://xsuds.argreenhouse.com/.

[15] J. Tsai, K. Fang, H. Chen, and Y. Bi, "A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging", *IEEE Trans. on Software Engineering*, Vol. 16, No. 8, (August 1990), pp. 897 - 916.

[16] A. Waheed, D. Rover, J. Hollingsworth, "Modeling and Evaluating Design Alternatives for an On-Line Instrumentation System: A Case Study", *IEEE Trans. on Software Engineering*, Vol. 24, No. 6, (June 1998), pp. 451 - 470.

[17] N. Wilde and M. Scully, "Software Reconnaissance: Mapping Program Features to Code", *Journal of Software Maintenance: Research and Practice*, Vol. 7, (1995) pp. 49-62.

[18] N. Wilde and C. Casey, "Early Field Experience with the Software Reconnaissance Technique for Program Comprehension", *Proc. International Conference on Software Maintenance - 1996*, IEEE Computer Society, Los Alamitos, CA, November 1996, pp. 312 - 318.

[19] Norman Wilde and Dean Knudson, "Understanding Embedded Software Through Instrumentation: Preliminary Results from a Survey of Techniques", Report SERC-TR-85-F, Software Engineering Research Center, Purdue University, 1398 Dept. of Computer Science, West Lafayette, IN 47906, February, 1999. Available at http://www.cs.uwf.edu/~wilde/publications/TecRpt85F_ExSum.html.

[20] J. Yan, S. Sarukkai, P. Mehra, "Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit", *Software - Practice and Experience*, Vol. 25, No. 4, (April 1995), pp. 429 - 461.