Scaling New Heights

WODA 2003
ICSE Workshop on Dynamic Analysis

ICSE'03
International Conference on Software Engineering
Portland, Oregon
May 3-11, 2003

Workshop on Dynamic Analysis
(WODA 2003)

May 9, 2003

Portland, Oregon

An ICSE 2003 Workshop

Organized by:

Jonathan Cook, New Mexico State University
Michael Ernst, Massachusetts Institute of Technology

Table of Contents

Program Analysis: A Hierarchy

Andreas Zeller
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken, Germany
zeller@acm.org

Abstract

Program analysis tools are based on four reasoning techniques: (1) deduction from code to concrete runs, (2) observation of concrete runs, (3) induction from observations into abstractions, and (4) experimentation to find causes for specific effects. These techniques form a hierarchy, where each technique can make use of lower levels, and where each technique induces capabilities and limits of the associated tools.

1. Introduction

Reasoning about programs is a core activity of any programmer. To answer questions like "what can happen?", "what should happen?", "what did happen?", and "why did it happen?", programmers use four well-known reasoning techniques:

Deduction from an abstraction into the concrete—for instance, analyzing program code to deduce what can or cannot happen in concrete runs.

Observation of concrete events—e.g. tracing, monitoring or profiling a program run or using a debugger.

Induction for summarizing multiple observations into an abstraction—an invariant, for example, or some visualization.

Experimentation for isolating causes of given effects— e.g. narrowing down failure-inducing circumstances by systematic tests.

These reasoning techniques form a hierarchy (Figure 1), in which each "outer" technique can make use of "inner" techniques. For instance, experimentation uses induction, which again requires observation; on the other hand, deduction cannot make use of any later technique.

The interesting thing about this hierarchy is that the very same reasoning techniques are also the foundations of automated *program analysis* tools. In fact, each of the reasoning techniques induces a specific class of tools, its capabilities and its limits. This is the aim of this paper: to provide a rough classification of the numerous approaches in program analysis—especially in dynamic analysis—, to show their common benefits and limits, and to show up new research directions to overcome these limits.

2. Deduction

Deduction is reasoning from the general to the particular; it lies at the core of all reasoning techniques. In program analysis, deduction is used for reasoning from the program code (or other abstractions) to concrete runs—especially for deducing what can or cannot happen. These deductions take the form of mathematical proofs: If the abstraction is true, so are the deduced properties.

Since deduction does not require any knowledge about the concrete, it is not required that the program in question is actually executed—the program analysis is *static*. Static
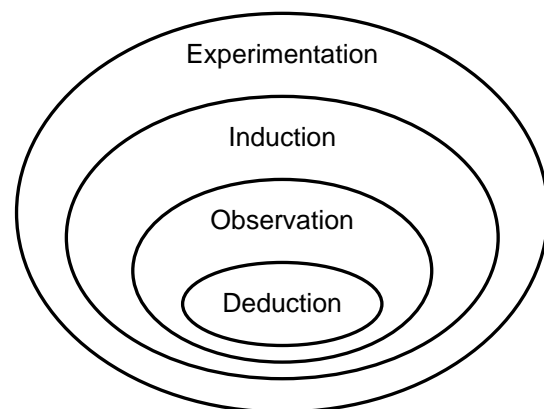

Figure 1. A hierarchy of reasoning techniques

program analysis was originally introduced in compiler optimization, where it deduces properties like

- Can this variable influence that other variable? (if not, one can parallelize their computation)

- Can this variable be used before it is assigned? (if not, there is probably an error)

- Is this code ever executed? (if not, it can be ignored)

Deduction techniques are helpful in program understanding, too—especially for computing *dependencies* between variables. A variable v' at a statement s' is dependent on a variable v at a statement s if altering v at s can alter the value of v' at s'; in other words, the value of v at s is a *potential cause* for v' at s'. By tracing back the dependencies of some variable v, one obtains a *slice* of the program—the set of all statements that could have influenced v [13, 14].

As an ongoing example, consider the following piece of C code. If p holds, a is assigned a value, which is then printed into the string buf.

```
3   char *format = "a = %d";
4   if (p)
5       a = compute_value();
6   sprintf(buf, format, a);
```

Let us assume that after executing this piece of code, we find that buf contains "a = 0". However, a is not supposed to be zero. What's the cause of "a = 0" in buf?

By deduction, we find that the string buf is set by the sprintf function which takes a as an argument; hence, buf depends on a at line 5. Likewise, a depends on p at line 4 (since altering p may alter a) and on the result of compute_value(). To find out why a is zero, we must trace back these dependencies in the slice. More important than the slice itself are the statements *not* included in the slice—e.g. a statement like c = d + e; The analysis proves that these cannot influence a or buf in any way; hence, they can be ignored for all further analysis.

Unfortunately, proving that executing some statement cannot influence a variable is difficult. Parallel or distributed execution, dynamic loading or reconfiguration of program code, unconstrained pointer arithmetic, or use of multiple programming languages are obstacles that are hard to handle in practice.

The biggest obstacle for deduction, though, is *obscure code:* If we cannot analyze some executed code, anything can happen. The sprintf function above, is typically part of the C runtime library and not necessarily available as source code. Only if we assume that sprintf works as expected can we ensure that buf depends on a.

3. Observation

Observation allows the programmer to inspect arbitrary aspects of an individual program run. Since an actual run is required, the associated techniques are called *dynamic*. Observation brings in actual *facts* of a program execution; unless the observation process is flawed, these facts cannot be denied.

For observing program runs, programmers and researchers have created a big number of tools, typically called "debuggers" because they are mainly used for debugging programs. A debugger allows to inspect states at arbitrary events of the execution; advanced tools allow a database-like querying of states and events [3, 12].

The programmer uses these tools to *compare* actual facts with expected facts—as deduced from an abstract description such as the program code. This comparison with expected facts can also be conducted automatically within the program run, using special *assertion* code that checks runtime invariants. Specific invariant checkers have been designed to detect illegal memory usage or array bound violations.

By combining slicing with observation, one obtains *dynamic slicing:* a slice that is valid for a specific execution only, and hence more precise than a slice that applies for all executions [1, 6, 11]. In principle, a dynamic slicing tool does not require source code as long as it can intercept all read/write accesses to program state and thus trace actual dependencies.

As an example of dynamic slicing, assume that after the execution of the code above, we find that buf contains "a = 0" and that p is true. Consequently, a dynamic slice tool can deduce from the code that the value of a can only stem from compute_value(); an earlier value of a cannot have any effect on buf (that is, unless a is being read in compute_value()).

Let's now introduce a little complexity: By observation, we also find that compute_value() returns a non-zero value. Yet, buf contains "a = 0". How can this be?

4. Induction

Induction is reasoning from the particular to the general. In program analysis, induction is used to *summarize* multiple program runs—e.g. a test suite or random testing—to some abstraction that holds for all considered program runs. In this context, a "program" may also be a piece of code that is invoked multiple times from within a program—that is, some function or loop body.

The most widespread program analysis tools that rely on induction are *coverage tools* that summarize the statement and branch coverage of multiple runs; such results can be easily visualized [10]. Most programming environments

7

support coverage tracing and summarizing. In program visualization, call traces and data accesses are frequently summarized [2].

On a higher abstraction level, *invariant detection* filters a set of possible abstractions against facts found in multiple runs. The remaining abstractions hold as invariants for all examined runs [4, 7]. This approach relies only on observation of the program state at specific events; hence, it is not limited by obscure code or other properties that make static analysis hard.

Both techniques can be used to detect *anomalies:* One trains the tool on a set of correct test runs to infer common properties. Failing runs can then be checked whether they violate these properties; these violations are likely to cause the failures.

As an example, let us assume that we execute the above C code under several random inputs, flagging an error each time `buf` contains `"a = 0"`. An invariant detector can then determine that, say, `a < 2054567 || a % 2 == 1` holds at line 6 for all runs where the error occurs. This is the common abstraction for all abnormal runs: `buf` contains `"a = 0"` whenever `a` is odd or smaller than 2,054,567. Obviously, something very strange is going on.

5. Experimentation

As in our C example, most problems in program understanding can be formulated as a search for *causes:* What is the cause for `buf` containing `"a = 0"`? It may be surprising that none of the techniques discussed so far is able to find an actual cause—or, more precisely, to *prove* that some aspect of a program is actually the cause for a specific behavior. To prove actual causality, one needs two experiments: one where cause and effect occur, and one where neither cause nor effect occur. The cause must precede the effect, and the cause must be a *minimal* difference between these experiments.

Searching for the actual cause thus requires a series of *experiments,* refining and rejecting hypotheses until a minimal difference—the actual cause—is isolated. This implies multiple program runs that are *controlled* by the reasoning process.

In our C example, our earlier induction step has already refined the cause in the program state: `a` is the cause for `buf` containing `"a = 0"`, because we can alter `a` such that `buf` has a different content. However, altering `a` in an experiment to, say, `2097153`, makes `buf` contain `"a = -2147483648"`. Would we consider this non-failing?

So, we decide that `a` is sane, and turn to the `sprintf` call. Assuming that `sprintf` works as specified, the only cause that can remain is the `format` string `"a = %d"` as

`sprintf` argument. Indeed, it turns out that `%d` is a format for integers, while `a` is declared as a floating-point value:

```
1 double a;
```

To verify that the format string is really the cause for `"a = 0"` in `buf`, we experimentally change the `format` variable from `"a = %d"` to `"a = %f"`. Our observation confirms that `buf` now has a sane value; this proves that the `format` string was indeed the cause for the failure.

Where do we obtain such alterations from? Obviously, a string like `format` can have an infinite number of possible contents. Finding the one format string that causes the bad `buf` content to turn into the correct one is left to the programmer; actually, this is part of writing a program that works as intended.

Nonetheless, even the search for causes can be automated—at least, if one has an alternate run where the effect does *not* occur. Our *delta debugging* approach can narrow down the initial difference between the two runs to the actual cause in program input [8] or program state [15]. Delta debugging creates artificial *intermediate* configurations that encompass only a part of the initial difference. Testing such configurations and assessing the outcome then allows to narrow down the actual cause.

Delta debugging has successfully isolated cause-effect chains from programs that so far had defied all kinds of deductive analysis, such as the GNU C compiler.

6. A Hierarchy of Program Analysis

By now, we have seen four techniques which are the foundation of program analysis tools. Each of these techniques induces a *class* of program analysis tools, defined by the *number of program runs* considered:

Deductive program analysis ("static analysis") generates findings *without executing* the program.

Observational program analysis generates findings from a *single execution* of the program.

Inductive program analysis generates findings from given *multiple executions* of the program.

Experimental program analysis generates findings from *multiple executions* of the program, where the executions are *controlled* by the tool.

As in Figure 1, these classes form a hierarchy where tools of each "outer" class may make use of the techniques in "inner" classes. Hence, dynamic slicing (observation) makes use of static slices (deduction); invariant detection (induction) relies on observation; delta debugging (experimentation) relies on observation and induction.

The classes also induce capabilities and limits:

- To determine causes, one needs experiments.

- To summarize findings, one needs induction over multiple runs.

- To find facts, one needs observation.

- And deduction, perhaps to some surprise, cannot tell any of these—simply because it abstracts from concrete program runs and thus runs the risk of abstracting away some relevant aspect.

However, deduction effectively proves what can and what cannot happen in the examined abstraction level; hence, it is an excellent guidance on what to observe, where to induce from and what to experiment.

7. Conclusion and Future Work

Program analysis tools can be classified into a hierarchy along the used reasoning techniques—deduction, observation, induction, and experimentation. Each class is defined by the used knowledge sources which impose capabilities and limits. This allows for a finer distinction of dynamic analysis techniques; names like observation, induction, or experimentation link directly to the techniques that programmers use in program comprehension.

While deduction and observation are quite well-understood, we have only yet begun to automate induction and experimentation techniques. Research in machine learning and data mining has produced a wealth of induction techniques. All of these can be applied to program runs in order to find patterns, rules, and anomalies—in runs and in code.

While induction works on a given set of program runs, we can use experimentation to gather more data from new, generated runs. The challenges here are when to use additional experimentation, how to generate runs that satisfy desired properties, and how to guide the experimentation process. The capability to design, run, and assess experiments automatically is unique to dynamic program analysis; we should make use of it.

Finally, program analysis can greatly benefit from further integration of "inner" tools and "outer" tools. Integrating experimentation with further inductive or deductive techniques is the main challenge in dynamic program analysis—and its greatest chance.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)*, volume 25(6) of *ACM SIGPLAN Notices*, pages 246–256, White Plains, New York, June 1990.

[2] W. de Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of java programs. In S. Diehl, editor, *Proc. of the International Dagstuhl Seminar on Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 163–175, Dagstuhl, Germany, May 2002. Springer-Verlag.

[3] M. Ducassé. Coca: An automated debugger for C. In *Proc. International Conference on Software Engineering (ICSE)*, pages 504–513, Los Angeles, California, May 1999.

[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001.

[5] W. G. Griswold, editor. *Proc. Tenth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-10)*, Charleston, South Carolina, Nov. 2002. ACM Press.

[6] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *Proc. ESEC/FSE'99 – 7th European Software Engineering Conference / 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 303–321, Toulouse, France, Sept. 1999. Springer-Verlag.

[7] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In ICSE 2002 [9], pages 291–302.

[8] R. Hildebrandt and A. Zeller. Simplifying failure-inducing input. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 135–145, Portland, Oregon, Aug. 2000.

[9] *Proc. International Conference on Software Engineering (ICSE)*, Orlando, Florida, May 2002.

[10] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In ICSE 2002 [9], pages 467–477.

[11] B. Korel and J. Laski. Dynamic slicing of computer programs. *The Journal of Systems and Software*, 13(3):187–195, Nov. 1990.

[12] R. Lencevicius. *Advanced Debugging Methods*. Kluwer Academic Publishers, Boston, 2000.

[13] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[14] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[15] A. Zeller. Isolating cause-effect chains from computer programs. In Griswold [5], pages 1–10.

Efficient Instrumentation for Performance Profiling

Edu Metz, Raimondas Lencevicius
Nokia Research Center
5 Wayside Road, Burlington, MA 01803, USA
Edu.Metz@nokia.com   Raimondas.Lencevicius@nokia.com

1. Introduction

Performance profiling consists of tracing a software system during execution and then analyzing the obtained traces. However, traces themselves affect the performance of the system distorting its execution [5]. Therefore, there is a need to minimize the effect of the tracing on the underlying system's performance. To achieve this, the trace set needs to be optimized according to the performance profiling problem being solved. Our position is that such minimization can be achieved only by adding the software trace design and implementation to the overall software development process. In such a process, the performance analyst supplies the knowledge of performance measurement requirements, while the software developer supplies the knowledge of the software. Both of these are needed for an optimal trace placement. The following sections expand on this position.

2. Performance profiling

Performance profiling is the means of determining where a software system spends its execution time. It uses trace instrumentation to gather event data. Various types of event information can be obtained with traces, such as component entry and exit, function calls, software execution states, message communication, resource usage, etc. However trace instrumentation comes at a cost — it impacts the performance of a software system [3][6]. For example, resource tracing is most of the time more intrusive then tracing function calls.

Not only does event tracing take some time, adding traces changes the behavior of the software system because of additional memory and I/O accesses [1]. In addition, in a real-time software system, instrumentation could possibly result in violation of real-time constraints and timing requirements. Trace instrumentation reduces the validity of performance profiling, so instrumentation has to be kept to a minimum.

2.1. Minimizing Performance Impact

There is a need to minimize the performance impact of trace instrumentation. To achieve this, we need to create efficient instrumentation. To instrument effectively, it is essential to know what events to monitor during execution of the software system and what information to collect when the event occurs. When instrumenting the software, it is essential to understand the purpose and goals of each trace and how it will affect the instrumented software component. From the performance profiling point of view, a "good" trace not only records the required event information; it also minimizes the impact on the system's performance, and does not violate any constraints and requirements.

In choosing the instrumentation granularity, it is important to address the trade-off between the amount of event information required and the performance impact of the trace instrumentation. For example: permanent OS traces in the scheduler report when a task switch occurs. These traces do not indicate if the task switch is due to preemption by a higher priority task or completion of the current running task. The duration of a task activity cannot be calculated based on OS scheduling traces only. It requires additional instrumentation. However, these additional traces will further impact the performance of the software system.

It should be noted that creating an efficient instrumentation does not eliminate the performance impact of trace instrumentation but rather tries to minimize the performance impact.

Let us summarize what we just talked about: efficient instrumentation for performance profiling imposes the following requirements:

- minimize the number of instrumentation points
- minimize the runtime overhead, and
- guarantee constraints and requirements.

2.2. Efficient Instrumentation

We need to establish instrumentation that meets the requirements outlined in the previous section. This can

be a complicated task, particularly in industry, where software development and performance profiling are often performed by different individuals each with their own set of skills and knowledge. Software developers have detailed knowledge of the software implementation. They understand the purpose of each instrumentation point and are able to assess the impact the instrumentation will have on the functional behavior of a software component. However, developers lack the understanding of what event data is needed. In addition, they may not be eager to insert event traces simply because they will not use them. On the other hand, performance analysts know what events need to be traced and understand what information needs to be recorded when an event occurs. However, performance analysts lack a detailed understanding of the software. We propose to draw upon the knowledge and skills software developers and performance analysts bring with them and use this knowledge to create efficient trace instrumentation.

To achieve this, we need to add trace instrumentation for performance profiling to the software development process. During the requirements phase the performance analyst should identify system-level performance requirements such as response time, throughput, and resource utilization, and start determining the events that need to be traced to check these requirements. For example, if the system level performance requirements state a maximum response time then the software's main entry and exit events (events e1 and e2 in Figure 1) need to be traced. However, it is not always possible to identify instrumentation points for all system level performance requirements during the requirements phase. For example, validation of resource utilization requirements requires knowledge of the software's execution states, which are not known until the design phase. Furthermore, only system level performance requirements are known during the specification phase. During the design phase, the performance analyst should identify lower level performance requirements such as messaging latency, interrupt response times, real-time deadlines, and time spent in the kernel. Next, the performance analyst should determine the events that need to be traced to check these requirements (for example, events e3 and e4 in Figure 1 as well as other events marked with black dots) and specify the event data that needs to be recorded when the event occurs. Typical events that need to be traced include: component entry and exit points, function calls, state transitions, message send and receive, and resource accesses. The developer then incorporates all the instrumentation requirements into the software design by identifying the corresponding instrumentation points. During implementation, the developer inserts traces at each event point, both manually and by activating (a subset of) permanent traces. The developer should plan to incrementally introduce the traces through iterations to

minimize the impact of the instrumentation code on software system operations. During this process, the performance analyst should provide guidance to the software developer on choosing the instrumentation granularity (e.g., trace events e5 and e8, but not events e6 and e7 in Figure 1).

Specification phase

Design phase

Implementation phase
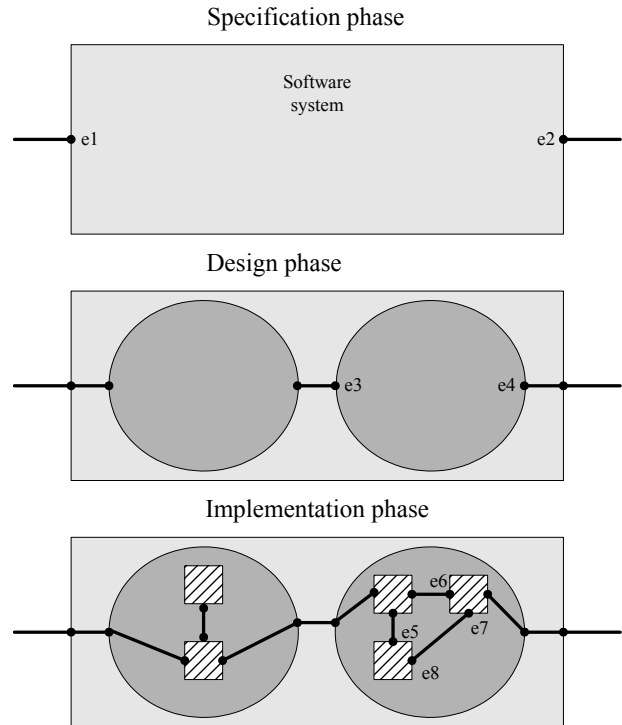
Figure 1: Trace design and implementation process

To illustrate this approach, let us look at an example. In mobile devices, power consumption is an important performance requirement [2]. The power consumption varies depending on the hardware resources used. During execution the software accesses hardware resources. These accesses need to be monitored to determine when a hardware resource is used, but should all access events be traced or is it enough to just trace enable and disable events? This question is best answered by the performance analyst. During the requirements phase the performance analyst identifies the power consumption requirements of the hardware resource. At design time, the performance analyst identifies the hardware access events that need to be traced to check the power consumption requirements. When tracing hardware access events in a mobile device it is very easy to violate real-time constraints and timing requirements. In addition, driver software of each hardware resource is unique. Instrumenting hardware drivers requires a detailed understanding of the software, and the developer is best suited for this task. During the design and implementation phase the developer

incorporates the instrumentation requirements set by the performance analyst into the driver software.

A good follow through by both the performance analyst and software developer is essential for the success of the proposed approach. For example: during the actual performance profiling phase, the performance analyst should relay any kind of trace instrumentation inefficiencies to the developer. The developer in turn should make the necessary instrumentation improvements and provide the performance analyst with an updated instrumented software build in a timely manner.

The approach to adding trace instrumentation for performance profiling to the software development process addresses the requirements outlined in section 2.1. In addition, this approach would yield some other incentives:

- allows for creating built in 'standardized' performance trace instrumentation, and
- provides formatting rules for performance event data.

Smith and Williams [4] proposes a systematic approach to software performance engineering. They focus on estimating the performance of a software system during each stage of the software development process. Our approach attempts to optimize the performance impact of trace instrumentation for performance profiling by adding the software trace design and implementation to the overall software development process.

3.   Summary

In this position paper, we described an approach to optimize trace instrumentation for performance profiling. The approach involves adding trace instrumentation for performance profiling to the software development process. It draws upon the knowledge and skills software developers and performance analysts bring with them — using this knowledge to create efficient trace instrumentation.

The proposed approach has the potential to decrease the number of instrumentation points. It would yield sufficient traces to profile the performance, yet it would not trace more event data than needed. In addition, the proposed approach would reduce the impact of trace instrumentation on software system performance.

4.   References

[1]   D. Konkin, G. Oster, R. Bunt, Exploiting Software Interfaces for Performance Measurement, *Proceedings of the 1st International Workshop on Software and Performance*, 1998, pp. 208–218.

[2]   R. Lencevicius, E. Metz, A. Ran, Software Validation using Power Profiles, *Proceedings of the 20th IASTED International Conference on Applied Informatics (AI),* 2002, pp. 143–148.

[3]   J. Moe, D. Carr, Understanding Distributed Systems via Execution Trace Data, 9*th International Workshop on Program Comprehension*, 2001, pp. 60–67.

[4]   C. U. Smith and L. Williams, *Performance solutions: A practical guide to creating responsive, scalable solutions,* Addison-Wesley, 2002.

[5]   D. Stewart, Measuring Execution Time and Real-Time Performance, *Embedded Systems Conference (ESC)*, 2001.

[6]   J. Vetter, D. Reed, Managing Performance Analysis with Dynamic Statistical Projection Pursuit, *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

Dynamic Analysis from the Bottom Up

Markus Mock
University of Pittsburgh
Department of Computer Science
6405 Sennott Square, Pittsburgh, PA 15260, USA
mock@cs.pitt.edu

Abstract

Changes in the way software is written and deployed today render static analysis increasingly ineffective. Unfortunately, this makes both traditional program optimization and software tools less useful. On the other hand, this also means that the role and importance of dynamic analysis is continuing to increase. In the future, we believe dynamic analysis will be successful both in program optimization and in software tools. One important ingredient to its success lies in efficient profiling methods. This paper looks at how this goal can be realized by exploiting already existing hardware mechanisms and possibly new ones. We believe that this will lead to software tools that are both effective and minimally invasive, easing their adoption by programmers.

1. Introduction

From its early beginnings, static analysis has been a huge success story. It is routinely used in optimizing compilers to ensure the correctness of code improving transformations. It is also commonly used in programming tools (e.g., smart editors) and software tools designed to facilitate the debugging and evolution of software, for instance, in program slicers. On the one hand, static analysis has been so successful because its use is unintrusive and does not require running the program or any other user intervention, and typically the user is completely unaware of its presence. On the other hand, to achieve practically useful results, typically the whole program, or large parts thereof have to be available to the analysis.

Unfortunately, this traditional model has been eroding over the last years thereby rendering traditional static analysis methods ever less effective. Since software is now routinely deployed as a collection of dynamically linked libraries, and more recently, also as Java bytecode that is delivered dynamically and on demand, static analysis in compilers and other programming tools knows less and less of the finally executing program. This forces it to make conservative assumptions that result in analysis results that are too imprecise to be useful either for program optimization or program "understanding" tasks.

While traditional static analysis is of limited effectiveness in these new dynamic software environments, *dynamic program analysis* [3] will play an increasingly important role to realize tasks that have become inefficient with static analysis alone. Moreover, dynamic analysis will enable new powerful techniques – both in optimization and program understanding – that are impossible to achieve with static analysis alone.

For some time now, dynamic (i.e., run-time) information has been used in optimizing compilers in the form of feedback-directed optimization where run-time information is used to aid the static program optimizer to make better optimization decisions – decisions, that would otherwise have to rely on static heuristics, which generally result in less effective optimization. More recently, run-time information has been exploited in dynamic compilation systems and just-in-time (JIT) compilers to which the complete program is available, which makes their analyses often quite successful [1].

While leveraging dynamic information in such systems has become quite popular, the use of dynamic analysis in software tools designed to assist the software engineer is still in its infancy. While the use of dynamic information in program optimization systems is always confined by the constraint of soundness – a potentially faster but possibly incorrect program has to be avoided –, tools designed to assist a human in a software engineering task are free of this restriction. Moreover, in many cases the results of a static analysis, although sound, may be considerably less useful than the potentially unsound result of a dynamic analysis, for instance, if it overwhelms the user with too much data.

For all the foregoing reasons, we believe that dynamic analysis algorithms, modeled after classical static analyses will be both important and useful in future software de-

velopment environments. Unconstrained by the yoke of soundness, dynamic analysis is likely going to be even more successful in software engineering applications than the promise it has already shown in run-time optimization. Crucial to the wider success of dynamic analysis, however, is the creation of efficient profiling methods to collect dynamic information unintrusively and with little performance overhead.

Therefore, we propose to design dynamic analysis systems "from the bottom up". Currently existing hardware mechanisms can be exploited to make the collection of run-time information more efficient. Software engineers interested in dynamic analysis should also work with hardware designers and compiler writers to participate in the design of new architectures that enable the efficient collection of data that can assist them in building more powerful and versatile dynamic analysis systems.

The rest of this paper is organized as follows: Section 2 discusses two future directions in the application of dynamic analysis. Section 3 looks at how to achieve efficient profiling methods as one essential ingredient in making dynamic analysis successful. Section 4 discusses related work and Section 5 concludes.

2. Future Directions in Dynamic Analysis

As the usefulness of static analyses decreases, dynamic analysis approaches are becoming more attractive. We see several interesting research directions for dynamic analysis in the coming years:

- research on how to effectively exploit run-time information to optimize programs;

- research on the application of dynamic analysis to improve software tools that assist programmers in the understanding, maintenance, and evolution of software; since such tools do not necessarily have to produce sound results this may be the "killer application" for dynamic analysis;

- research on the efficient collection of run-time information; this includes research into combined hardware-software approaches that will lower the cost of collecting run-time information.

In the following two sections, we will briefly discuss the first two items, which represent two broad application areas for dynamic analysis. In Section 3 we will then elaborate on the last point, which is fundamental for the wider success of dynamic analysis.

2.1 Program Optimization with Dynamic Analysis

To achieve good program performance, increasingly runtime information will be necessary to perform effective code-improving transformations. The fundamental constraint for program optimization, though, is soundness, which is at odds with the unsound nature of dynamic analysis. However, we believe that a symbiosis of static and dynamic analysis will not only be effective but in fact crucial for the success of program optimization of future software systems.

Results of static analysis are always conservative approximations of actual run-time behavior; when programs are only partially known, this problem is exacerbated because worst case assumptions have to be made for all unknown code parts. On the other hand, program properties may in fact be true in most, if not all, runs despite the inability of static analyses to demonstrate this. For instance, [9] has shown that the statically computed sets of potential pointer targets in pointer dereferences in C programs are several orders of magnitude larger than the number of actually observed targets. Consequently, optimizing compilers are often not able to allocate variables to registers because of aliases through pointer accesses,[1] even though those accesses at run-time never or almost never overwrite the variable's value.

Fortunately, static and dynamic analysis may be combined in this case to improve what can be done with static analysis alone. One approach consists of generating multiple code versions, one, in which code is optimized aggressively assuming that no aliasing occurs even though the static analysis is not able to ascertain this. The decision when this specialized code should be generated would be based on a dynamic analysis that checks at program execution whether aliasing does occur. A run-time check would then be inserted in the code to select the correct code version and ensure soundness.

Similarly, other program properties that are usually inferred by static program analysis, might be observed at run time. Static analysis would then be used to generate appropriate run-time checks to ensure the soundness of program transformations that depend on the correctness of those properties. Investigating what properties are both useful and efficiently derivable by dynamic analysis, is an interesting research area for the combination of static and dynamic analysis as well as the exploration of synergies arising from that combination.

---

[1]Alternatively, if they are allocated to registers, after every possibly aliased write through a pointer, the register value has to be reloaded, which may neutralize the benefit of register allocating the variable.

2.2 Improving Software Tools with Dynamic Analysis

Whereas dynamic analysis will generally have to be complemented by static analysis to be applicable in program optimization, software engineering tools may enjoy the benefits of dynamic analysis in many cases even without supporting static analysis. As has been observed by several researchers, in many cases unsound information may be just as useful or even more useful than sound information in software engineering applications.

The key to the usefulness of dynamic analysis again is the (increasing) imprecision of static analyses. While static analyses may provide a sound picture of program properties. this picture may be too complex to be useful in practice. As an example, consider again pointer analysis. A static points-to analysis[2] may compute several hundreds or even thousands of potential pointer targets for a dereference. When a user wants to understand what are in fact the feasible targets, a points-to set of that size will be too large to be examined completely. Moreover, the static analysis does not provide any insights into which of those targets are more likely than others to show up in practice.

On the other hand, a dynamic points-to analysis [9] shows only those points-to targets that have actually occurred at run time. While this set may not be sound, i.e., miss some targets that may in fact be feasible, since dynamic points-to sets are typically very small, they can be much more useful because the enable the programmer to focus on definitely feasible targets, which in addition, may be prioritized by the frequency of occurrence, so that any subsequent task can be focused to examine the more important (more frequent, or more likely) analysis results first. Dynamic pointer information has been used, for instance, to improve program slicing [8].

3 Dynamic Analysis from the Bottom Up: Achieving Efficient Profiling

One of the fundamental challenges for the success of dynamic program analysis lies in the creation of instrumentation and profiling infrastructures that enable the efficient collection of run-time information. Current approaches typically result in significant program slowdowns [9, 5] so that they are confined to offline use. They are also not invisible to the user and typically additional effort is required to integrate them with current software tools. While this may be acceptable when the result is directly consumed by the user of a dynamic analysis tool, when the dynamically derived information is subsequently used to transform a pro-

gram for instance, faster turnaround times will significantly enhance the usability of tools based on dynamic analysis. Moreover, if dynamic analysis can be performed with minimal overheads during normal program executions, it may become routine and not require any additional effort from software engineers to tap into the generated information.

In our opinion, one particularly promising approach to reduce profiling overhead lies in the collaboration with computer architects. Processor designers dispose of more hardware resources than ever so that it is not unreasonable to expect that additional structures to support efficient dynamic analysis may be placed on chips if they provide a significant enhancement of functionality. Moreover, very simply hardware structures may suffice and can potentially make a big difference in performance. For example, the addition of hardware data watchpoints in modern processors (for example the Intel Pentium), enables a debugger such as gdb to monitor all memory accesses to a particular variable (or set of variables) without noticeable performance degradation on the program. When such hardware support is not present, monitoring the contents of a variable becomes often prohibitively expensive – typical software implementations based on trapping after every instruction, result in slowdowns of a factor of 100.

The additional hardware required to support data watchpoints, on the other hand, is minimal. Similarly, for many of the properties we are interested in dynamic program analysis it may be possible to achieve big performance improvements with simple hardware support. Current processors already have many hardware performance counters, which are used in profiling for program performance. Maybe future architectures will have "analysis counters" to assist software engineers in building fast dynamic analysis tools. If we can show that such support is useful for the software community as a whole, we should have a good case for their realization in silicon.

The following sections will look at potential mechanisms to aid in two particular dynamic analysis tasks: points-to profiling and invariant detection.

3.1 Example One: Points-To Profiling

Maintaining a mapping from the current addresses of local and heap-allocated variables to their compile-time names accounts for the major part of the cost of points-to profiling.[3] If the compiler could simply load the monitored addresses into a hardware table and all load and store instructions would automatically be checked against this table (simultaneously updating the associated access statistics), points-to profiling would only add a small amount of extra

---

[2]A points-to analysis computes for each pointer dereference in a program the set of potential targets accessed by the dereference, called the *points-to set* of the dereference.

[3]The addresses of local variables usually change with each invocation and multiple addresses are usually associated with the same memory allocation site.

work (the initialization of the address table at procedure entry and at each malloc site). In current software implementations, for every load and store instruction tens or hundreds of instructions have to be executed resulting in slowdowns of one to two orders of magnitude [9].

Some current processors, e.g., the Intel Itanium, already support a similar, though more limited hardware structure (the *ALAT* table [7]), which, however, cannot be directly loaded by the compiler (it is manipulated indirectly through special load instructions used for optimization). Therefore, it appears not unreasonable to assume that a more general mechanism similar to the one described above, may eventually be implemented in hardware.

3.2 Example Two: Invariant Detection

Another field where compiler and architecture support can be used to improve the applicability of dynamic analysis is invariant detection. In the Daikon [5] system, invariants are detected offline after a profiling urn of an instrumented program. Obviously, for dynamically updated software this two-phased approach does not directly work since the program needs to be re-instrumented as it is running. Moreover, it may actually be desirable to detect some invariants as the program is running, for instance when invariants represent security-relevant properties.

Arnold and Ryder [2] present an approach to reduce the cost of instrumented code by providing a mechanism to dynamically enable and disable the profiling of selected program parts. Their approach could be combined with Daikon in a run-time system that would automatically instrument dynamically changing code. The dynamically updated code could then be gradually profiled to detect its (local) invariants and as soon as invariants stabilize, profiling would be disabled until the next software update. This would enable invariant detection while a system is running, at potentially very little overhead, so that invariant detection could remain in place even in deployed software.

This would make exciting new applications possible. For instance, software could be shipped with previously detected or specified invariants. As the system runs, these invariants could be compared against those detected in the field. Discrepancies, which might indicate, for example, insufficient testing, could then be reported back to the developer to either correct the software or the invariants.

4. Related Work

PREfix [4] was one of the first tools that tried to overcome the imprecision of static program analysis by a systematic exploration of program execution paths along which certain program properties were checked. It was shown to be very effective in detecting program errors that were not detectable by static analysis. Ernst [5] has focused on detecting likely program invariants, which can then be used to reason about programs or in error detection. The DIDUCE system [6] uses dynamic program analysis to detect unusual program states which are likely to indicate program bugs.

5. Conclusions

Due to changes in the way software is written and deployed today, the effectiveness of static analysis is decreasing. Therefore the importance of dynamic analysis will continue to increase. Consequently, improving the usability of dynamic analysis tools by making them less intrusive and more efficient is one of the main challenges for dynamic analysis researchers today. By designing dynamic analyses from the bottom up, and in collaboration with compiler writers and computer architects, we believe that efficiency and ease of use will be achieved and make dynamic analysis a standard feature of future software systems.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, USA, Oct. 2000.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 168–179. ACM Press, 2001.

[3] T. Ball. The concept of dynamic analysis. In *ESEC / SIGSOFT FSE*, pages 216–234, 1999.

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, June 2000.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[6] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM Press, 2002.

[7] Intel Corporation. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, 2002.

[8] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. In *Proceedings of the 10th ACM International Symposium on the Foundations of Software Engineering*, Charleston, SC, Nov. 2002.

[9] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72, Snowbird, UT, USA, June 2001.

Exploiting Synergy Between Testing and Inferred Partial Specifications

Tao Xie   David Notkin

Department of Computer Science & Engineering, University of Washington
{taoxie, notkin}@cs.washington.edu

Abstract

The specifications of a program can be dynamically inferred from its executions, or equivalently, from the program plus a test suite. A deficient test suite or a subset of a sufficient test suite may not help to infer generalizable program properties. But the partial specifications inferred from the test suite constitute a summary proxy for the test execution history. When a new test is executed on the program, a violation of a previously inferred specification indicates the need for a potential test augmentation. Developers can inspect the test and the violated specification to make a decision whether to add the new test to the existing test suite after equipping the test with an oracle. By selectively augmenting the existing test suite, the quality of the inferred specifications in the next cycle can be improved while avoiding noisy data such as illegal inputs. To experiment with this approach, we integrated the use of Daikon (a dynamic invariant detection tool) and Jtest (a commercial Java unit testing tool). This paper presents several techniques to exploit the synergy between testing and inferred partial specifications in unit test data selection.

1. Introduction

Given that specifications play an important role in a variety of software engineering tasks and that the specifications are often absent from a program, dynamically inferring program specifications from its executions is a useful technique [3]. The output of the dynamic specification inference has been used to aid program evolution in general [3] and program refactoring in particular [7]. Most of the applications can achieve better results if the inferred specifications are closer to the oracle specifications. Like other dynamic analysis techniques, the dynamic specification inference is also constrained by the quality of the test suite for the program. Usually it is unlikely that the inferred properties are true over all possible executions. When properly applied, static verification tools can filter out false positives in the inferred specifications [8].

Different from previous applications that use the final inferred specifications from all the available tests, two recent approaches have begun to use the intermediate partial specifications inferred from a subset. Both are based on the fact that the inferred specifications may change when new tests are added to the existing test suite. The first, called the operational difference (OD) technique, makes use of the differences in inferred specifications between test executions to generate, augment, and minimize the test suites [5]. The second, as implemented in the tool DIDUCE, can continuously check a program's behavior against the incrementally inferred partial specifications during the run(s), and produce a report of all violations detected along the way [4]. This can help detect bugs and track down the root causes. It is noteworthy that "partial specification" also carries the denotation that the specification is not complete or accurate in terms of an oracle specification. Thus there is a convergence of the two meanings when the specifications inferred from the whole test suite are used to approximate the oracle specification.

In this research, we further exploit the synergy between testing and inferred partial specifications. All available tests in this context are a small size of the existing unit test suite plus a large size of the automatically generated unit tests. The purpose is to tackle the problem of selecting automatically generated tests to augment the existing unit test suite. Violations of the inferred partial specifications from the existing unit test suite can help this unit test data selection. Moreover, selectively augmenting the existing test suite can prevent introducing noisy data, e.g. illegal inputs, from negatively affecting the specification inference.

2. Background

The "test first" principle, as advocated by Extreme Programming (XP) development process [1], requires unit tests to be constructed and maintained before, during, and after the source code is written. Developers need to manually generate the test inputs and oracles based on the

requirements in mind or in documentation. They need to decide whether enough test cases have been written to cover the features in their code thoroughly. Some commercial tools for Java unit testing, e.g. ParaSoft Jtest [10], attempt to fill the "holes" left by the execution of the manually generated unit tests. These tools can automatically generate a large number of unit tests to exercise the program. However, there are two main issues in automatic unit test generation. First, there are no test oracles for these automatically generated tests unless developers write down some formal specifications or runtime assertions [2]. Second, only a relatively small size of automatically generated tests can be added to the existing unit test suite. This is because the unit test suite needs to be maintainable, as is advocated by the XP approach [1].

Two main unit test selection methods are available. In white box testing (e.g., the residual structural coverage [11]), users select tests that provide new structural coverage unachieved by the existing test suite. In black box testing, the operational difference (OD) technique is applicable in augmenting a test suite [5]. However, the OD technique for this unit test augmentation problem might select a relatively large set of tests because the specification generator's statistical tests usually require multiple executions before outputting a specification clause. Additionally, OD requires frequent generation of specifications, and the existing dynamic specification generation is computationally expensive. Therefore, instead of using OD in the unit test selection, we adopt a specification violation approach similar to DIDUCE [4].

Our approach is implemented by integrating Daikon and Jtest. Daikon [3], a dynamic invariant detection tool, is used to infer specifications from program executions of test suites. The probability limit for justifying invariants is set by Daikon users. The probability is Daikon's estimate of how likely the invariant is to occur by chance. It ranges from 0 to 100 with a default value of 1 . Smaller values yield stronger filtering. Daikon includes a MergeESC tool, which inserts inferred specifications to the code as ESC Java annotations [12]. ParaSoft Jtest [10], on the other hand, is a commercial Java unit testing tool, which automatically generates unit test data for a Java class. It instruments and compiles the code that contains Java Design-by-Contract (DbC) comments, then automatically checks at runtime whether the specified contracts are violated. We modified MergeESC to enable Daikon to insert the inferred specifications into the code as DbC comments. Since ESC Java has better expressiveness than Jtest's DbC, a perl script is written to filter out the specifications whose annotations cannot be supported by Jtest's DbC. After being fed with a Java class annotated with DbC comments, Jtest uses them to automatically create and execute test cases and then verify whether a class behaves as expected. It suppresses any

problems found for the test inputs that violate the preconditions of the class under test. But it still reports precondition violations for those methods called indirectly from outside the class. Note that DIDUCE tool reports all precondition violations [4]. By default, Jtest tests each method by generating arguments for them and calling them independently. In other words, Jtest basically tries the calling sequences of length 1 by default. Tool users can set the length of calling sequences in the range of 1 to 3. If a calling sequence of length 3 is specified, Jtest first tries all calling sequences of length 1 followed by all those of length 2 and 3 sequentially.

3. Specification Violation Approach

This section describes the specification violation approach. Section 3.1 introduces the basic technique of the approach. Section 3.2 presents the precondition guard removal technique to improve the effectiveness of the basic technique. Section 3.3 describes the iterative process of applying these techniques. A preliminary experiment is conducted on a Java class of the bounded stack that is used to store unique elements of integer [13]. Detailed experimental results for this example are described in [14].
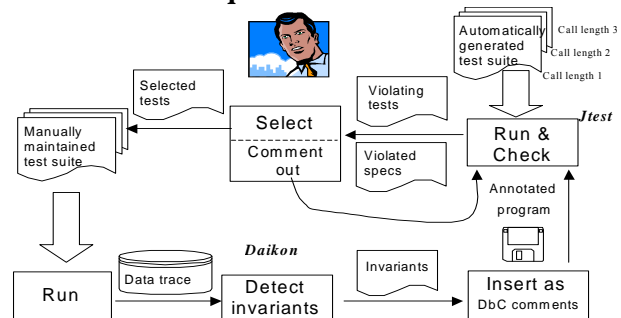
3.1. Basic Technique

Figure 1. An overview of the basic technique

In our approach, partial specifications are inferred from program executions of the existing unit test suite by using Daikon (Figure 1). The partial specifications are inserted into the code as DbC comments. The resulting code is fed to Jtest. Initially, Jtest's calling sequence length is set to 1 and Jtest is run to automatically generate and execute test data. When a certain number of specification violations have occurred before Jtest exhausts its testing repository, it stops generating test data and reports specification violations. For each reported specification violation, i.e., the violated specification and the violating test, developers inspect them to decide whether to equip the test with an oracle and add it to the existing test suite. Then developers disable each violated specification by

18

commenting them out and rerun Jtest repeating the above procedure until no specification violations are reported. The whole process is iteratively applied by setting the length of calling sequences as 2 and subsequently 3.

The rationale behind the basic technique is that if a new test violates an inferred partial specification, it is likely that this test exercises a new feature of the program uncovered by the existing test suite. This technique guarantees that the new test does not overlap with any others from the existing test suite in terms of the violated specification. In addition, the violating tests have a relatively high probability of exposing faults in the code if there are any. It is because that running the existing test suite on the code exhibits the normal behavior reflected by the inferred specifications and the violating tests might make the code exhibit the abnormal behavior.

The symptoms of specification violations can be that the boolean value of a specification predicate is false or exceptions are thrown. In order for the inferred specifications to be violated, we set the probability limit to be 100 . The specification violations indicate deficiencies of the existing test suite. However, some violations might not be very helpful for the unit test selection. For example, the existing test suite for the stack implementation only push the integer element of 2 or 3 into the stack and thus one of the inferred specifications is that the stack element is 2 or 3. The automatically generated tests that push the element of 1 into the stack violate this specification. Since the element of 1 is not so different than 2 or 3 for the purpose of testing this stack implementation, developers might not select the violating test to the existing test suite.

3.2. Precondition Guard Removal

In our basic technique, when the existing test suite is deficient, the inferred preconditions might be so restrictive as to filter out those legal test data inputs in Jtest test data generation and execution. This over-restrictiveness of preconditions also makes static verification of inferred specifications less effective [8]. Even if a static verifier could confirm an inferred post-condition specification given some over-restrictive preconditions, it is hard to tell whether it is generalizable to the actual preconditions.

To assure better quality of the unit under test, we need to exercise the unit under more circumstances than what is constrained by the inferred preconditions. Before the code that is annotated with DbC comments is fed to Jtest, all precondition comments are removed. In the preliminary experiment, we observed that precondition guard removal techniques reported more violations and exposed more faults than the basic technique (Section 3.1). Indeed, removing precondition guards produces more false positives by allowing some illegal inputs. et the tool

only reports those illegal inputs that cause postcondition or invariant violations.

3.3. Iterations

After the new test augmentations using the 3.1 and 3.2 techniques, all the violating tests with legal inputs, whether selected or unselected, can be further run together with the existing ones to infer new specifications. Although those unselected violating tests with legal inputs might not exercise any interesting new features, running them in the specification inference can relax the violated specifications to reduce the false positives in the next iteration. The same process described in Section 3.1 and 3.2 is repeated until there are no specification violations or no test data selected from the violating tests. In the preliminary experiment, most of the specification violations were observed in the first iteration, and all specification violations were observed before the third iteration.

4. Effect of Inferred Specifications on Test Generation

In previous sections, we showed that the inferred specifications can be used to select unit test data and improve the specification quality. Furthermore, we observed that the inferred specifications also had an effect on Jtest's automatic test generation. As is described in Jtest's manual [6], if the code has preconditions, Jtest tries to find inputs that satisfy all of them. If the code has postconditions, Jtest creates test cases that verify whether the code satisfies these conditions. If the code has invariants, Jtest creates test cases that try to make them fail. The preliminary experiment showed that preconditions have greater impacts on Jtest's test generation than either postconditions or invariants. Sometimes Jtest, equipped with specifications, could automatically generate tests that achieve better code coverage than the one without specifications. For the test length of two, the former Jtest automatically generated more tests for the stack implementation than the latter one. It suggests that inferred specifications are able to guide Jtest to generate better tests.

5. Concluding Remarks

In sum, selecting automatically generated tests to augment the existing unit test suite is an important step in the unit testing practice. Inferred partial specifications act as a proxy for the existing test execution history. A new test that violates an inferred specification is a good candidate for developers to inspect for test data selection. The violating test also has a high probability to expose

faults in the code. Instead of considering the test augmentation as a one-time phase, it should be considered as a frequent activity in software evolution, if not as frequent as regression unit testing. When a program is changed, the specifications inferred from the same unit test suite might change as well, giving rights to possible test violations. Tool-assisted unit test augmentation can be a means to evolving unit tests and assuring better unit quality. Moreover, augmenting unit test suite in a controlled way can lead to better quality of inferred specifications. In future work, we plan to apply the specification violation techniques in connecting system testing and unit testing. Specifications are to be inferred from system testing and specification violations by the generated unit tests are used to guide unit test data selection. Also, the partial specifications inferred from testing done by component providers are to be delivered as component metadata [9], which will aid component users to perform test augmentations. Finally, we plan to apply the specification violation techniques in other kinds of inferred specifications, e.g. sequencing constraints or protocols.

6. Acknowledgement

7. References

[1] K. Beck. Extreme programming explained. Addison-Wesley, 2000.

[2] . Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Proceedings of 16th European Conference Object-Oriented Programming (ECOOP), 2002, pp. 231-255.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, vol. 27, no. 2, Feb. 2001, pp. 1-25.

[4] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In Proceedings of the International Conference on Software Engineering, May 2002, pp. 291-301.

[5] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In Proceedings of the International Conference on Software Engineering, (Portland, Oregon), May 6-8, 2003.

[6] Jtest manuals version 4.5. Parasoft Corporation, October 23, 2002.

[7] . Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In Proceedings of ICSM 2001, November, 2001, pp. 736-743.

[8] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC Java. In Proceedings of RV'01, First Workshop on Runtime Verification, (Paris, France), July 23, 2001.

[9] A. Orso, M. J. Harrold, and D. Rosenblum. Component metadata for software engineering tasks, In Proceedings of the 2nd International Workshop on Engineering Distributed Objects, November 2000, pp. 129-144.

[10] ParaSoft Corportation. http: www.parasoft.com

[11] C. Pavlopoulou and M. oung. Residual test coverage monitoring. In Proceedings of ICSE 1999, pp. 277-284.

[12] K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. ESC Java user's manual. Technical Report 2000-002, Compaq Systems Research Center, Palo Alto, California, Oct 12, 2000.

[13] P. D. Stotts, M. Lindsey, A. Antley. An informal formal method for systematic JUnit test case generation. XP Agile Universe 2002, pp 131-143.

[14] T. Xie and D. Notkin. Exploiting synergy between testing and inferred partial specifications, University of Washington Department of Computer Science and Engineering technical report UW-CSE-03-04-02, (Seattle, WA), April 2003.

Generating Test Data for Dynamically Discovering Likely Program Invariants

Neelam Gupta

Department of Computer Science
The University of Arizona
Tucson, AZ 85721
ngupta@cs.arizona.edu

Abstract

Dynamic detection of program invariants is emerging as an important research area with many challenging problems. As with any dynamic approach, the accuracy of dynamic discovery of likely program invariants depends on the quality of test cases used to detect invariants. Therefore, generating suitable test cases that support accurate detection of program invariants is crucial to the dynamic approach for invariant detection.

In this paper, we explore new directions in using the existing test data generation techniques to improve the accuracy of dynamically detected program invariants. First we discuss the augmentation of existing test suites to improve the accuracy of dynamically discovered invariants. The augmentation of the test suite may be done prior to running the dynamic analysis if the variables and expressions whose values will be monitored at runtime are known in advance. On the other hand, the dynamic analysis may be run first using an available test suite to obtain an initial guess of program invariants. These guessed invariants may then be used to generate test cases to augment the test suite. We also propose the use of existing test data generation techniques in improving the accuracy of invariants guessed using an already available test suite.

Keywords - *Test data generation, path testing, program invariants, dynamic analysis, execution traces.*

1 Introduction

Dynamic detection of program invariants is an emerging area of research with many challenging problems [3, 4]. The accuracy of dynamically discovered invariants critically depends upon the test suite used for detection of invariants. One parameter of the test suite that can be loosely related to the accuracy of dynamic detection of invariants is the size of the test suite. However, not all large test suites can be expected to be equally effective in accurate detection of invariants due to varying degree of structural coverage obtained. Thus, it is crucial to conduct research on what properties make a test suite suitable for dynamic invariant detection.

In prior work [3, 4], randomly generated and grammar generated test suites have been used for invariant detection. Randomly generated test suites have poor coverage and are most effective at highly peculiar bugs [10]. In the experiments reported in [4], the randomly generated test suites failed to execute many portions of a program. These randomly generated test suites did not detect many of the invariants that were detected using hand-crafted input cases. The experiments using randomly generated test suites from a grammar describing valid inputs detected more invariants than completely randomly generated test suites. However, generating test cases using grammar rules is a black box approach to test case generation and in general can fail to cover a significant part of the implementation.

In this paper we explore new research directions in generation of test cases to support dynamic invariant detection. We discuss the augmentation of existing test suites to improve the accuracy of dynamically discovered invariants. The augmentation of the test suite may be done prior to running the dynamic analysis if the variables and expressions, whose values will be monitored at runtime, are known in advance. On the other hand, the

dynamic analysis may be run first using an available test suite to guess program invariants. These guessed invariants may then be used to generate test cases to augment the test suite. We also propose the use of existing test data generation techniques to improve the accuracy of dynamically discovered likely invariants.

The organization of the paper is as follows. We discuss the background work in test data generation and dynamic detection of program invariants in section 2. In section 3, we propose new research directions to improve the accuracy of dynamically discovered invariants. Finally, we summarize the contributions of this paper and our future work.

2   Background

Test Data Generation Problem   We consider the problem of generating input data that forces execution through a given path in a program. Symbolic evaluation [1, 2] and program execution based approaches [7, 8, 5, 11] have been proposed for generating test data for a given path in a program. The problem of test data generation for a given path is defined as follows.

> **Problem Statement:** *Given a program path P which is traversed for certain evaluations (true or false) of branch predicates $BP_1, BP_2 \cdots BP_n$ along P, generate a program input $I = (i_1, i_2..., i_m)$ in the input domain of the program that causes the branch predicates to evaluate such that P is traversed.*

The selection of paths for which the test input needs to be generated depends upon the testing strategy. For example, if the testing strategy is to ensure coverage of all branches in the program, the test paths are selected so that each branch is exercised by at least one test path among those selected.

Dynamic Invariant Detection.   We consider the approach to dynamic discovery of invariants presented in [3, 4]. In this approach the invariants are dynamically detected from program traces that capture the variable values at program points of interest. The user runs the target program over a test suite to create execution traces of the program. An invariant detector determines which properties hold over both explicit variables and other expressions. Variable and expressions for which these properties hold over the traces, and also satisfy other tests such as being statistically justified, not being over unrelated variables and not being implied by other invariants, are reported as likely in-

variants. The set of likely invariants reported depends on the test suite used to discover invariants.

3   Test Data Generation for Dynamic Invariant Detection

In this paper we explore the relationship between the test data generation problem and dynamic discovery of program invariants. First we illustrate that the test suites satisfying the statement and branch coverage criteria may not be good enough for accurate detection of program invariants. We propose new approaches to to augment these test suites with additional test cases that can help in improving the accuracy of detected invariants. Second we illustrate the use of test data generation techniques in improving the accuracy of detected invariants.

3.1   Augmenting a Test Suite for Invariant Detection

We first illustrate the limitations of using existing structural coverage test suites for dynamic discovery of program invariants and propose how these test suites may be augmented with additional test cases to overcome these limitations.

```
0:    int funcEx(int x, y)
1:    {
P1:       if (x > 0)
2:           a=3;
3:           c=6;
4:       else
5:           a=3;
6:           c=9;
7:       endif
P2:      if (y > 0)
8:           b=4;
9:           d=2;
10:      else
11:          b=3;
12:          d=1;
13:      endif
14:   /* Monitored Property: (a*b == c*d) */
15:          printf(" a*b == c*d")

16:      ⋮
17:   }
```

Figure 1. An example code segment

Let us consider the code segment shown in Figure 1. Let the expression *(a*b == c*d)* in line 15 represent a property to be monitored during every ex-

ecution of this code segment. The code segment has been instrumented so that the value of this expression is written into every execution trace for this code segment. Let us say the test suite T_1 consists of the following two input cases.

$T_1 = \{(x = 5, y = 2), (x = -5, y = -1)\}$

Note that executing the code segment in Figure 1 with test cases in T_1 executes every statement in this code segment. In addition, every branch outcome of the two branch predicates $P1$ and $P2$ are executed by this test suite. Also note that every *definition-use* pair in this code segment is also exercised by this test suite. The property tested in line 15 will also hold for this test suite T_1. But it is easy to see that this property does not hold for the test case $(x = 5, y = -1)$. This simple example illustrates that code coverage (each statement executed at least once by some test case) and even branch coverage (each branch outcome is evaluated at least once by some test case) are very weak criteria for the test suite to be adequate for dynamic invariant detection.

However, the above example provides insight into the limitations of using coverage based test suites for detecting invariant properties at different points in the programs. These test suites are designed to test structural coverage of the program and may not contain test cases that are specifically helpful in verification of properties being monitored for invariant discovery. What is needed is the augmentation of these test suites with test cases specific to the properties being monitored.

In the above example, we need test cases for all possible combinations of branch outcomes by which the program execution can reach the critical point where the property of interest is being monitored. But in general, the number of paths reaching the critical point may be unbounded due to the presence of loops. So the crucial problem is *how to identify the important paths reaching the critical point* so that augmenting the structural coverage test suites with the test cases for these paths gives higher confidence in the value of the property being monitored during execution.

One approach we propose is to select the paths that exercise different *definition-use* pairs that are *live* at the critical point where the invariant property is being monitored. However, in order to compute the *live definition-use* pairs at the critical point, we need to know the expression or the variable that is being monitored at this point. If the explicit variables and other expressions whose properties are collected in the executions traces are available in

advance, then the *live definition-use* pairs for these variables and expressions can be computed.

On the other hand, if the explicit variables and expressions whose properties are to be monitored are not available in advance, then runtime analysis [3, 4] can be used to discover likely invariants with an existing structural coverage test suite. The *live definition-use* pairs for the discovered likely invariants (at the relevant program points) can then be used to guide the selection of paths important for verification of these discovered invariants. The test inputs for these paths can be generated and the structural coverage test suite can then be augmented with these test inputs. Now if the runtime analysis [3, 4] is done with the augmented test suite, it is expected that some of the spurious invariants that were reported earlier with the structural coverage test suite may not be reported any more. This is because the augmented test suite contains test cases specific to verification of those likely invariant properties that were reported earlier by the structural coverage test suite. The subset of the properties reported (from among those reported with the coverage test suite) by the augmented test suite is expected to be more accurate than the original set of likely invariants reported with the structural coverage test suite. We are currently exploring the effectiveness of this approach in our ongoing research. In the next section, we illustrate a different dimension of the relationship between the test data generation problem and the accuracy of reported program invariants.

3.2 Formulating Invariant Detection Problem as a Test Data Generation Problem

We propose to formulate the invariant detection problem as a data generation problem to improve the accuracy of dynamically discovered invariants. We illustrate this with the example in Figure 1. Let us replace line 15 in the code segment shown in Figure 1 by lines $P3$, 15, 16 and 17 shown in Figure 2.

We call the new branch predicate $P3$ introduced in the code segment in Figure 3 as the *invariant checking predicate*. Let us consider the problem of generating test data to execute the branch denoted by the line $P3$ followed by line 17, i.e., the *false* branch outcome of predicate $P3$. Now, *if test data can be generated for the false branch of an invariant checking predicate, then the corresponding property does not hold irrespective of the information collected from the execution traces using the already available test*

```
0:    int funcEx(int x, y)
1:    {
P1:       if (x > 0)

             .
             .
             .

7:        endif
P2:       if (y > 0)

             .
             .
             .

13:       endif
14:   /* Monitored Property: (a*b == c*d) */
P3:       if (a*b == c*d)
15:           printf("Property holds")
16:       else
17:           printf("Not an invariant")

18:          .
             .
19:   }
```

Figure 2. Modified example code segment

suites. As can be seen for the example in Figure 2, test data for the *false* outcome of $P3$ will be easily generated by program execution based techniques in [9, 12].

The above example illustrates an important application of the test data generation techniques in support of dynamic invariant detection. If test data can be generated to exercise the false branch of an invariant checking predicate, then the corresponding guessed invariant must be discarded. This is because this test input serves as a counterexample to this guessed invariant. Although, in general it is undecidable whether there exists an input to execute a given path in an arbitrary program, techniques [1, 2, 7, 8, 5, 11] have been developed for automatic generation of test data for a given path in a program. Different test data generation techniques have different strengths and the difficulty of test data generation for a path depends on the complexity and interdependence of branch predicates along the path. However, whenever test data generation techniques can generate an input exercising the *false* branch of an invariant checking predicate, the accuracy of the reported invariants can be significantly improved.

4   Conclusions and Future Work

In this paper we have provided insight into the relationship between test cases used for detecting invariants and the accuracy of invariant properties thus detected. We have proposed approaches for augmenting test suites for accurate detection of invariants. We are currently exploring these ap-

proaches for their effectiveness in accurate discovery of program invariants. We have also proposed the use of test data generation techniques to improve the accuracy of dynamically discovered program invariants.

References

[1] L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pages 215-222, September 1976.

[2] R.A. DeMillo and A.J. Offutt, "Constraint-based Automatic Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 17, No. 9, pages 900-910, September 1991.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering,* vol. 27, no. 2, Feb. 2001, pp. 1-25.

[4] M. D. Ernst. "Dynamically Discovering Likely Program Invariants," *Ph.D. dissertation,* University of Washington Department of Computer Science and Engineering, (Seattle, Washington), Aug. 2000.

[5] M.J. Gallagher and V.L. Narsimhan, "ADTEST: A Test Data Generation Suite for Ada Software Systems," *IEEE Transactions on Software Engineering*, Vol. 23, No. 8, pages 473-484, August 1997.

[6] A. Gotlieb, B. Botella, and M. Rueher, "Automatic Test Data Generation using Constraint Solving Techniques," *International Symposium on Software Testing and Analysis*, 1998.

[7] N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated Test Data Generation using An Iterative Relaxation Method" *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering (FSE-6)*, pages 231-244, Orlando, Florida, November 1998.

[8] N. Gupta, A. P. Mathur, and M. L. Soffa, "UNA Based Iterative Test Data Generation and its Evaluation," *14th IEEE International Conference on Automated Software Engineering(ASE'99),* pages 224-232, Cocoa Beach, Florida, October 1999.

[9] N. Gupta, A. P. Mathur, M. L. Soffa, "Generating Test Data for Branch Coverage", *15th IEEE International Conference on Automated Software Engineering (ASE'00),* Grenoble, France, September 2000.

[10] D. Hamlet, "Random Testing," *Encyclopedia of Software Engg.,* 1994.

[11] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, pages 870-879, August 1990.

[12] B. Korel, A Dynamic Approach of Test Data Generation. In *Conference on Software Maintenance,* pages 311-317, San Diego, CA, November 1990.

Static and dynamic analysis: synergy and duality

Michael D. Ernst

MIT Lab for Computer Science
Cambridge, MA 02139 USA
mernst@lcs.mit.edu

Abstract

This paper presents two sets of observations relating static and dynamic analysis. The first concerns synergies between static and dynamic analysis. Wherever one is utilized, the other may also be applied, often in a complementary way, and existing analyses should inspire different approaches to the same problem. Furthermore, existing static and dynamic analyses often have very similar structure and technical approaches. The second observation is that some static and dynamic approaches are similar in that each considers, and generalizes from, a subset of all possible executions.

Researchers need to develop new analyses that complement existing ones. More importantly, researchers need to erase the boundaries between static and dynamic analysis and create unified analyses that can operate in either mode, or in a mode that blends the strengths of both approaches.

1. Background

This section briefly reviews some facts about traditional static and dynamic analyses, to set the stage for the rest of the paper.

Static and dynamic analyses arose from different communities and evolved along parallel but separate tracks. Traditionally, they have been viewed as separate domains, with practitioners or researchers specializing in one or the other. Furthermore, each has been considered ill-suited for the tasks at which the other excels. This paper argues that the difference is smaller than it appears and that certain of these distinctions are unnecessary and counterproductive.

Static analysis examines program code and reasons over all possible behaviors that might arise at run time. Compiler optimizations are standard static analyses. Typically, static analysis is conservative and sound. Soundness guarantees that analysis results are an accurate description of the program's behavior, no matter on what inputs or in what environment the program is run. Conservatism means reporting weaker properties than may actually be true; the weak properties are guaranteed to be true, preserving soundness, but may not be strong enough to be useful. For instance, given a function f, the statement "f returns a non-negative value" is weaker (but easier to establish) than the statement "f returns the absolute value of its argument." A conservative analysis might report the former, or the even weaker property that f returns a number.

Static analysis operates by building a model of the state of the program, then determining how the program reacts to this state. Because there are many possible executions, the analysis must keep track of multiple different possible states. It is usually not reasonable to consider every possible run-time state of the program; for example, there may be arbitrarily many different user inputs or states of the run-time heap. Therefore, static analyses usually use an abstracted model of program state that loses some information, but which is more compact and easier to manipulate than a higher-fidelity model would be. In order to maintain soundness, the analysis must produce a result that would be true no matter the value of the abstracted-away state components. As a result, the analysis output may be less precise (more approximate, more conservative) than the best results that are in the grammar of the analysis.

Dynamic analysis operates by executing a program and observing the executions. Testing and profiling are a standard dynamic analyses. Dynamic analysis is precise because no approximation or abstraction need be done: the analysis can examine the actual, exact run-time behavior of the program. There is little or no uncertainty in what control flow paths were taken, what values were computed, how much memory was consumed, how long the program took to execute, or other quantities of interest. Dynamic analysis can be as fast as program execution. Some static analyses run quite fast, but in general, obtaining accurate results entails a great deal of computation and long waits, especially when analyzing large programs. Furthermore, certain problems, such as pointer or alias analysis, remain beyond the state of the art; even exponential-time algorithms do not always produce sufficiently precise results. By contrast, determining at run time whether two pointers are aliased requires a single machine cycle to compare the two pointers (somewhat more, if relations among multiple pointers are desired).

The disadvantage of dynamic analysis is that its results may not generalize to future executions. There is no guarantee that the test suite over which the program was run (that is, the set of inputs for which execution of the program was observed) is characteristic of all possible program executions. Applications that require correct inputs (such as semantics-preserving code transformations) are unable to use the results of a typical dynamic analysis, just as applications that require precise inputs are unable to use the results of a typical static analysis. Whereas the chief challenge of building a static analysis is choosing a good abstraction function, the chief challenge of performing a good dynamic analysis is selecting a representative set of test cases (inputs to the program being analyzed). (Efficiency concerns affect both types of analysis.) A well-selected test suite can reveal properties of the program or of its execution context; failing that, a dynamic analysis indicates properties of the test suite itself, but it can be difficult to know whether a particular property is a test suite artifact or a true program property.

Unsound dynamic analysis has been traditionally denigrated by the programming languages community. Semantics-preserving program transformations such as compiler optimizations require correct information about program semantics. However, unsoundness is useful in many other circumstances. Dynamic analysis can be used even in situations where program semantics (but not perfect program semantics) are required. More importantly, humans are remarkably resilient to partially incorrect information [10], and are not hindered by its presence among (a sufficient quantity of) valuable information. Since in most domains human time is far more important than CPU time, it is a better focus for researchers. As a result, and because of its significant successes, dynamic analysis is gaining credibility.

2. Static and dynamic analysis: synergies

As noted in Section 1, static and dynamic analysis have complementary strengths and weaknesses. Static analysis is conservative and sound: the results may be weaker than desirable, but they are guaranteed to generalize to future executions. Dynamic analysis is efficient and precise: it does not require costly analyses, though it does require selection of test suites, and it gives highly detailed results regarding those test suites.

The two approaches can be applied to a single problem, producing results that are useful in different contexts. For instance, both are used for program verification. Static analysis is typically used for proofs of correctness, type safety, or other properties. Dynamic analysis demonstrates the presence (not the absence) of errors and increases confidence in a system.

This section considers the use of static and dynamic analysis in tandem, to complement and support one another. First, static and dynamic analyses enhance each other via pre- or post-processing. Second, existing static and dynamic analyses can suggest new analyses. Third, static and dynamic analyses should be combined into a hybrid analysis.

2.1. Performing both static and dynamic analysis

Static or dynamic analyses can enhance one another by providing information that would otherwise be unavailable. Performing first one analysis, then the other (and perhaps iterating) is more powerful than performing either one in isolation. Alternately, different analyses can collect different varieties of information for which they are best suited.

This well-known synergy has been and continues to be exploited by researchers and practitioners alike. As one simple example, profile-directed compilation [1] uses hints about frequently executed procedures or code paths, or commonly observed values or types, to transform code. The transformation is meaning-preserving, and it improves performance under the observed conditions but may degrade it in dissimilar conditions (the correct results will still be computed, only consuming more time, memory, or power). As another example, static analysis can obviate the collection of certain information by guaranteeing that collecting a smaller amount of information is adequate; this makes dynamic analysis more efficient or accurate.

2.2. Inspiring analogous analyses

Both static and dynamic analysis can always be applied to a particular program, though possibly at different cost, and their results have different properties. Whenever only one of the analyses exists, it makes sense to investigate the other, which may be able to use the same technical approach. In many cases, both approaches have already been implemented by different parties.

One simple example is static and dynamic slicing [14]. Slicing indicates which parts of a program (may) have contributed to the value computed at, or the execution of, a particular program expression or statement. Slicing can operate statically, dynamically, or both.

As a more substantive example, Purify [8] and LCLint [6] are tools for detecting memory leaks and uses of dead storage. (Each has capabilities missing from the other, but this discussion considers only the intersection of their capabilities.) Purify performs a run-time check, essentially by use of tagged memory. Each byte of memory used by the program is allocated a 2-bit state code indicating whether that memory is unallocated, uninitialized, or initialized; at each memory access, the memory's state is checked and/or updated by instructions that Purify inserts in the executable.

LCLint operates statically, checking user-supplied annotations that indicate assumptions. It performs a dataflow analysis whose abstract state contains includes definedness and allocation state; each program operation has particular requirements on its inputs and produces certain results. The rules and abstract states used by Purify and LCLint are essentially identical: they perform the same analysis, Purify dynamically and LCLint statically.

As another example, consider program specifications, which are formal mathematical abstractions of program behavior. When used to verify behavior, the standard static technique is theorem proving, which typically requires human interaction. The dynamic analog of theorem-proving is the `assert` statement, which verifies the truth of a particular formula at run time. Specifications are best written by the designer before implementation commences. When specifications are synthesized after the fact, the typical approach is a static one that proceeds by examining the program text. This task is sometimes done automatically with the assistance of heuristics, but very frequently it is done by hand. The dynamic analog to writing down a specification is generating one automatically by dynamic detection of likely invariants [4, 5]. The invariant detection technique postulates potential invariants, tests them over program executions, and then prunes them via static analysis, statistical tests, heuristics, and other techniques. As a result, its output is often close to the ideal (over its grammar) that a perfect static analysis or human would produce [11].

Dynamic invariant detection was invented as a direct result of considering the duality between dynamic and static analysis. There existed static analyses that could generate specifications (or formulas syntactically identical to specifications, if the term "specification" is reserved for human-produced formulas), but no dynamic analyses existed. (Dynamic techniques for other varieties of specifications already existed [2].) This led to a new technique that has since been applied to refactoring, bug detection, fault isolation, test suite improvement, verification, theorem-proving, detection of component incompatibilities, and other tasks. Other researchers would be well advised to look for other missing analyses, in order to inspire development of new analyses by comparison with their existing analogs. Where just one (static or dynamic) analysis exists, the other is likely to be advantageous.

2.3. Hybrid static-dynamic analysis

Presently, tool users must select between static and dynamic analysis. (Section 2.1's noted cooperative strategies use one analysis as a prepass for the other, but the overall output is that of the final analysis.) In some cases, one or the other analysis is perfectly appropriate. However, in other cases, users may prefer not to be forced to choose between the two approaches.

A better alternative is to create new, hybrid analyses that combine static and dynamic analyses. Such an analysis would sacrifice a small amount of the soundness of static analysis and a small amount of the accuracy of dynamic analysis to obtain new techniques whose properties are better-suited for particular uses than either purely static or purely dynamic analyses.

The hybrid analyses would replace the (large) gap between static and dynamic analysis with a continuum. Users would select a particular analysis fitted to their needs: they would, in a principled way, turn the knob between soundness and precision. It seems unlikely that one extreme or the other is always the appropriate choice: users or system builders should be able to find the "sweet spot" for their application. Indeed, different analyses (both static and dynamic) already use different amounts of processing power to produce results of differing precision. This could be a starting point for the work. Another starting point could be use of only a subset of all available static information, much as already practiced by some tools [12, 7]. A third starting point is an observation about the duality of static and dynamic analysis, noted immediately below in Section 3. One potential barrier is different treatments (optimistic vs. conservative) of unseen executions.

3. Static and dynamic analysis: duals

Static and dynamic analysis are typically seen as distinct and competing approaches with fundamentally different the techniques and technical machinery. (Section 2.2 noted that in some cases, the underlying analyses are quite similar.) This section argues that the two types of analysis are not as different as they may appear; rather, they are duals that make many of the same tradeoffs.

The key observation is that both static and dynamic analysis are able to directly consider only a subset of program executions. Generalization from those executions is the source of unsoundness in dynamic analysis and imprecision in static analysis.

A dynamic analysis need not be unsound. A sound dynamic analysis observes every possible execution of a program. If a test suite contains every possible input (and every possible environmental interaction), then the results are guaranteed to hold regardless of how the program is used. This simple goal is unattainable: nontrivial programs usually have infinitely many possible executions, and only a relatively small (even if absolutely large) set of them can be considered before exhausting the testing budget (in time, money, or patience). Researchers have devised a number of techniques for using partial test suites or for selection of partial test suites [13]. These techniques are of interest solely as efficiency tweaks to an algorithm that works perfectly in theory but exhausts resources in practice.

A static analysis need not be approximate. A perfectly precise static analysis considers every possible execution of a program, maintaining, for each execution, the program's full state (or, rather, all possible states). This is not typically feasible, because there are infinitely many possible executions and the state of the program is extremely large. Researchers have devised many abstractions, primarily of state but also of executions, that permit them to consider a smaller state space or a smaller number of executions, reducing the problem to one that can often be solved on today's computers. The abstractions are of interest solely as efficiency tweaks to an algorithm that works perfectly in theory [3] but exhausts resources in practice.

Both dynamic and static analyses consider only a subset of all possible executions, but that subset is chosen differently. (Executions not in the set may be dealt with differently, as well. In particular, the unobserved executions may be treated conservatively and pessimistically or may be treated optimistically, which often means simply ignoring them. This distinction between sound and unsound analysis is important but is omitted for reasons of space and because it is orthogonal to the main point.)

The set of executions considered by a dynamic analysis is exactly those that appear in the test suite or that were observed during execution. This set is very easy to enumerate and may characterize a particular environment well; however, the set may be difficult to formalize in mathematical notation. The set of executions considered by a static analysis is those that induce executions of a certain variety. For instance, the k-limiting [9] abstraction considers in detail only the executions that create data structures with pointer-directed paths of length no more than k; another popular abstraction considers only executions that traverse each loop either zero or one times [6, 7].

Each of the descriptions is simpler in some respects and more complicated in others. Given a data-structure-centric description like those used for static analysis, it is difficult to know what executions induce the data structures or whether particular programs or execution environments will suffer degradation of analysis results. Given a set of inputs or executions, analysis is required to understand what parts of a program are exercised, and in what ways.

Recognition of this duality — both analyses consider a subset of executions — should make it easier to translate approaches from one domain to the other and to combine static and dynamic analyses, or at least to a better understanding of the gap between them.

4. Conclusion

This paper has listed some widely-recognized distinctions between static and dynamic analysis, notably soundness versus precision. It noted ways that static and dynamic analysis can interact: by augmenting one another, by inspiring new analyses, and by creating hybrid analyses that combine them. Some of these seem to have been overlooked by previous authors. Finally, it noted a duality between static and dynamic analysis, both of which consider (differently-specified) subsets of program executions. We encourage other researchers to join us in bringing these research ideas to fruition.

References

[1] B. Calder, P. Feller, and A. Eustace. Value profiling. In *MICRO-97*, pages 259–269, Dec. 1–3, 1997.

[2] J. E. Cook and A. L. Wolf. Event-based detection of concurrency. In *FSE*, pages 35–45, Nov. 1998.

[3] P. M. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *ACM Symposium on Artificial Intelligence and Programming Languages*, pages 1–12, Aug. 1977.

[4] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, U. Wash. Dept. of Comp. Sci. & Eng., Seattle, Washington, Aug. 2000.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, Feb. 2001.

[6] D. Evans. Static detection of dynamic memory errors. In *PLDI*, pages 44–53, May 21–24, 1996.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[8] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Winter 1992 USENIX Conference*, pages 125–138, Jan. 1992.

[9] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[10] G. C. Murphy, D. Notkin, and E. S.-C. Lan. An empirical study of static call graph extractors. In *ICSE*, pages 90–99, Mar. 1996.

[11] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 232–242, July 2002.

[12] PREfix/Enterprise. www.intrinsa.com, 1999.

[13] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE TSE*, 24(6):401–419, June 1998.

[14] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

Improving Design Pattern Instance Recognition by Dynamic Analysis[*]

Lothar Wendehals
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany

lowende@upb.de

Abstract

Design pattern instance recognition is often done by static analysis, thus approaches are limited to the recognition of static parts of design patterns. The dynamic behavior of patterns is disregarded and leads to lots of false positives during recognition. This paper presents an approach to combine the advantages of static and dynamic analyses to overcome this problem and improve the design pattern instance recognition.

1. Motivation

Reverse engineering large industrial legacy systems is hard work. They consist of several thousand or up to million lines of code and often lack of documentation. The systems have grown over several years and were developed by different programmers with different programming styles.

Design recovery, which means extracting design documents from source code, is a way to assist the reengineer understanding and maintaining those systems. As a basis for design documentation design patterns first presented by Gamma et al. [4] are suitable. By recognizing instances of design patterns in the system's source code, the implicit design may be recovered and documented. Further enhancements can then be applied to the system.

Most approaches to design recovery use static analysis techniques on the system's source code [1, 6, 7, 12]. Some of them are text-search tools based on regular expressions. Other approaches use graph representations of the source code, such as control flow or data flow graphs or even abstract syntax trees.

In object-oriented languages those static analyses are not sufficient. Polymorphism and dynamic method binding prevent the correct analysis of method invocations that are essential to recover patterns with behavioral aspects such as the *Chain of Responsibility* pattern [4] depicted in Figure 1.

Some parts of a *Chain of Responsibility* pattern such as the inheritance between the abstract class Handler and their concrete children classes or the self-association successor of the class Handler can be found by static analyzing techniques. Method calls such as the delegation between a Handler object and its successor can be found statically, but the concrete invoked method and the concrete object the method is invoked on can only be analyzed during runtime.

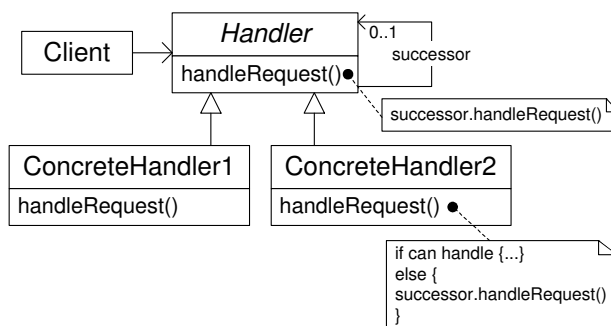Thus a precise recognition of design pattern instances with

Figure 1: *Chain of Responsibility* **pattern**

dynamic behavior requires dynamic analysis. A complete reengineering process based on dynamic analysis only is not appropriate, because static parts of design patterns can be identified easier in static analysis. So a smart combination of static and dynamic analysis is desirable.

The combined reengineering process starts with the static analysis of the source code. As a result of this first part of the process a set of pattern instance candidates is produced. This set is the input for the dynamic analysis part of the process. It reduces the search space for the dynamic analysis. During runtime of the program pattern instance candidates only have to be investigated.

In the following an overview of our pattern-based design recovery process is presented. An example for a pattern instance is then given to clarify the limitations of static analysis. To lift this restrictions dynamic analysis is added to design pattern instance recognition based on static analysis. The paper closes with related work and some conclusions.

2. Pattern-based Design Recovery

In our approach described in [8, 9] we use an abstract syntax graph (ASG) representation of the source code. This ASG is produced by parsing the source code. It contains static information about classes, attributes, methods including method bodies and inheritance. Our approach is not bound to any particular programming language. As a case study we analyzed software systems written in Java.

We use additional nodes to enrich the ASG with information gathered during analyses. Those nodes added to the ASG are called annotations. They are linked to the nodes in the ASG that have to be annotated with information.

A tool-based design recovery needs formalized rules for the analysis. We developed a graphical rule definition language based on graph-rewrite-rules with a left and a right side. Each pattern that should be searched for is defined by the left side of such a graph-rewrite-rule. The right side of the rule consists of the pattern together with the annotation node that has to be added to the ASG. By successfully applying these rules to the ASG, pattern instances are recovered. The information of a found pattern instance is stored by the annotation node linked to the ASG elements that are participating in the pattern's instance.

By defining new rules, existing rules can be reused. Simple rules may be combined to new more complex and more abstract rules. As a result a pattern rule catalog is formed where rules depend on each other.

To support reverse engineering tasks where up to million lines of code are analyzed we developed a highly scalable design recovery process. We showed that our approach is applicable to real life software systems such as the Java Abstract Windowing Toolkit (AWT) [11] with more than 140.000 lines of code [8, 9].

3. Static Analysis

Pattern-based design recovery is a deductive analysis problem where patterns, or rules, are repeatedly applied to a representation of the source code to arrive at the most complete characterization of the code permitted by the rules. Pure deductive analysis algorithms typically apply the rules involved level by level - bottom-up - according to their natural hierarchy. Results from other researchers, such as [13] and [10], suggest that a reverse engineering tool providing fully automatic analysis based on this approach cannot scale for larger software systems.

We developed a combined bottom-up and top-down strategy. The rules in the pattern rule catalog are sorted by their natural dependency hierarchy. The analysis starts in bottom-up mode with rules at the lowest level which are rules that do not depend on others. After successfully applying such a rule, consequent rules at the next level will be triggered. If any rule depends on precondition rules that have not yet applied, the strategy switches into the top-down mode. After evaluating all preconditions the strategy changes back to bottom-up mode. The whole analysis algorithm which ensures a highly scalable process can be found in [8].

In our ASG representation of the source code method bodies are also contained as mentioned before. This enables our static analysis to analyze parts of the dynamic behavior of methods. The existence of method calls can be identified but dynamic method binding and polymorphism prevents to identify the actual called method and the actual object the method is invoked on. It can only be a first indication of dynamic behavior.

Figure 2 depicts an instance of a *Chain of Responsibility* pattern shown in Figure 1. This example shows a part of a model for a graphical user interface. There is an abstract class GUIElement that implements a multiple self-reference children. Concrete subclasses of this abstract class are a Window, a Panel and a Button. They override a method from their superclass. The dotted line of the inheritance relation denotes an indirect inheritance. So there are other classes in between the inheritance hierarchy.

Suppose a pattern rule is defined to identify a *Chain of Responsibility* pattern instance as shown in Figure 1. The
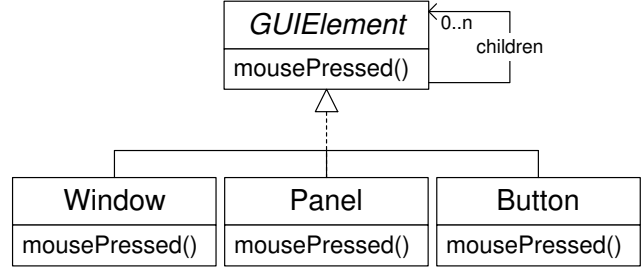

Figure 2: Concrete instance of a *Chain of Responsibility* pattern

source code to be reengineered contains a *Chain of Responsibility* pattern instance as depicted in Figure 2. During static analysis there are some elements of uncertainty that prevent an exact matching of this pattern instance.

The multiple self-reference of the abstract superclass GUIElement is different from the single reference successor of the *Chain of Responsibility* pattern. This could be a counter indicator for a *Chain of Responsibility* pattern instance, because a chain element has always only one successor. Another uncertainty derives from the indirect inheritance hierarchy. The original pattern describes a direct inheritance between the abstract handler and its concrete handlers. Furthermore the method call delegation from a handler to its successor can not be identified exactly. A method call from a handler to another handler can be statically identified, but it is not for sure that this call is forwarded in a chain of objects.

This leads only to an inexact match. There are two ways to handle this match. Firstly, this match can be discarded, because it is different from the original defined pattern. Secondly, it could be accepted as a pattern instance candidate with a low certainty of being a correct pattern instance. This certainty is expressed as a fuzzy value. In [9] we describe how to handle inexact pattern matches by fuzzy values.

The result of the overall static analysis is a set of pattern instance candidates each rated by a fuzzy value. For some of the candidates the certainty (fuzzy value) that they are actual pattern instances is not very high because of dynamic behavior that can not be analyzed statically as stated before. Some of them may even be false positives. Dynamic analysis can help to make these results more precise.

The analysis restricted to the candidates reduces the input for dynamic analysis. To further reduce the search space, our static analysis process provides the analysis of method bodies as part of the ASG. Structural information about method bodies such as a method call within a loop can be used for refining the rules. This reduces not only the number of candidates but also the number of methods that have to be investigated by dynamic analysis. Methods that are probably not participating in the pattern can be separated from those that are relevant to the pattern.

4. Dynamic Analysis

The design patterns descriptions used by Gamma et al. [4] are informal in most parts, for example the motivation, applicability, consequences and implementation. More formal parts of a pattern description are the structure and sometimes the collaboration parts. The collaboration parts often

contain UML sequence diagrams with typical behavior of the pattern constituents. Figure 3 shows such a sequence diagram for the *Chain of Responsibility* pattern. Those descriptions of the dynamic behavior of patterns can be used by the reverse engineer to formally define rules for tool-based design recovery.
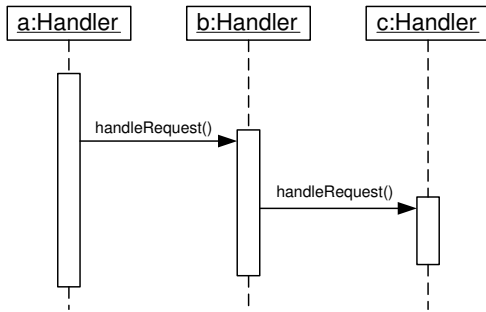
Figure 3: Sequence diagram pattern for a *Chain of Responsibility*

For each pattern with dynamic behavior a pattern for a UML sequence diagram is added to the pattern's rule. It describes typical sequences of method calls between objects that participate in the pattern. The diagrams can only be samples for object interaction. Figure 3 for example only shows three objects acting as a *Chain of Responsibility*. Actual chain of responsibilities may consist of more than three objects.

Reengineering a program often aims at changing or adding features. The program's part to be reengineered is therefore precisely defined. So the execution of the program for dynamic analysis can be restricted to those parts. The execution has to be done manually by the reengineer.

Information will be gathered during program execution by debugging the program. Basic functionality of debuggers allow to set breakpoints and record method traces. For each pattern-relevant method from candidate classes breakpoints are set. The pattern-relevant methods can be found by static analysis as mentioned before. So object information and their method traces are recorded during runtime. These information are stored as an attributed call graph and form the data for the pattern instance recognition.

The procedure of the dynamic analysis is analog to static analysis. After generating a call graph by executing the program - which corresponds to parsing the source code into an ASG in static analysis - the gathered information has to be analyzed. The sequence diagrams are defined as graph-rewrite-rules just like the static part of a pattern rule. The matching of the sequence diagrams can now be done by applying their graph-rewrite-rules to the attributed call graph, which again corresponds to applying the static pattern rules to the ASG. Finally the results of both analyses - static and dynamic - are rated by fuzzy values.

During runtime of the program there could be multiple different object sets that are instances of one pattern instance candidate. For example a *Chain of Responsibility*-pattern used in a program can be instantiated multiple times during runtime. For each of these sets object type information and method traces will be recorded. Polymorphism and dynamic method binding enables method traces of the sets to differ significantly from each other, even if they are instances of the same pattern instance candidate. In our example there could be object sets instantiated from the same *Chain of Responsibility* instance where the objects are different concrete handlers. Method traces from those sets would be different.
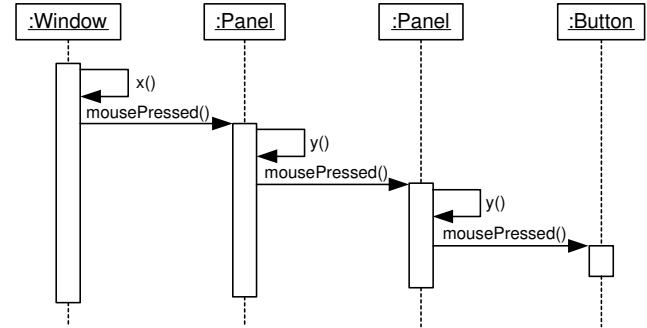
Figure 4: Method trace from a candidate object set

Figure 4 depicts an example for a object set and its method trace. These objects are instances from the class diagram of Figure 2. This object set is therefore an instance of a pattern instance candidate. There is a mouse pressed event that is delegated from a Window object to the responsible Button object. Some method calls as x() or y() may have been recorded, too. Others may have been suppressed, as method calls to different objects that were not investigated.

The matching between the pattern sequence diagram and the method trace can only be inexact. There are three objects in the pattern sequence diagram depicted in Figure 3 delegating the handleRequest() method call to their successor. This situation can be found in the method trace of Figure 4 if naming is not considered. There is one additional object and there are additional method calls that do not match any method call in the pattern. So a matching can be found but it is ambiguous and inexact. The grade of ambiguity and inexactness has to be rated for each object set and its method trace. The rating is expressed by a fuzzy value within a range between 0 and 1 like in static analysis.

Both results from static and dynamic analysis are then presented to the reengineer and has to be interpreted. There are three cases that have to be considered for each pattern instance candidate.

Firstly, there were no object set that was instantiated from the candidate during program execution. So there is only a result from static analysis. That means the features executed do not use the program's part the pattern instance candidate belongs to. Therefore the reengineer is not interested in that program's part and design and the results from static analysis can be ignored.

Secondly, there are one or more object sets with their method traces for one pattern instance candidate. In this case the fuzzy values from all object sets are combined to three values: the minimum, the average and the maximum fuzzy value.

Suppose in our example there are five object sets instantiated from the pattern instance candidate of Figure 2. Four of these sets have a fuzzy value of 0.9 and one set has a fuzzy value of 0.4. The average fuzzy value is 0.8. The static analysis result for the given example is a certainty of 0.6 of being an actual *Chain of Responsibility* pattern instance. The maximum fuzzy value from dynamic analysis confirms

this assumption. The minimum fuzzy value is a contraindication, but the average value shows that most of the fuzzy values confirm the assumption. In the case of a low average fuzzy value the result would indicate a false positive.

Thirdly, there are object sets for a pattern instance candidate, but the given sequence diagram could not be matched to the call graph. All three fuzzy values - minimum, average and maximum - will be null. This indicates that the pattern instance candidate from static analysis can be clearly identified as a false positive.

5. Related Work

Heuzeroth et al. [5] combine as well static as dynamic analysis to detect interaction patterns. They approach is similar to that presented in this paper. The source code is represented by an abstract syntax tree (AST). Static patterns are described as relations over AST node objects. The computed relations are input to the dynamic analysis. Dynamic patterns are described by protocols over a set of events. The relations as well as the protocols have to be implemented manually, which means implementing the algorithms to calculate the relations and to calculate the match of protocols. This restricts the usability of the approach because of the complicated maintenance, adaption and creation of patterns. Furthermore the approach for static analysis is limited in recognizing implementation variants of patterns. This leads either to lots of false positives or to missing pattern instances. Lots of false positives in static analysis cause then a higher complexity in dynamic analysis.

Eisenbarth et al. [2] combine static and dynamic analysis as well. Their approach helps the reengineer identifying components used for certain features. In contrast to the presented approach Eisenbarth et al. use dynamic analysis to reduce the search space for static analysis. Scenarios for all features that have to be located in the code are chosen for the program's execution. Concept analysis is then performed to identify relationships between scenarios and subprograms. These results are used in static analysis which is done by slicing techniques and manual inspection. So the search space should be small for static analysis.

6. Conclusions

An approach is presented to use dynamic program analysis to confirm results from static analysis. The static analysis as described in this paper is already implemented in our CASE tool FUJABA [3]. The implementation of the dynamic analysis is current work. Pattern rule specification and matching for dynamic analysis will be realized by graph-rewrite-rules as in static analysis. The inference algorithm [8] can therefore be reused.

We introduced the notion of fuzziness into our static analysis to rate pattern instance candidates [9]. This approach is used for the rating in dynamic analysis, too. The combination of both results from static and dynamic analysis is presented to the reengineer for each pattern instance candidate. The dynamic analysis results confirm or discard the static analysis result. Thus, the combination is a good criterion for the reliability of the results.

7. References

[1] G. Antoniol, R. Fiutem, and L. Christoforetti. Design pattern recovery in object-oriented software. In *Proc.* of the 6th *International Workshop on Program Comprehension (IWPC), Ischia, Italy*, pages 153–160. IEEE Computer Society Press, June 1998.

[2] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM 2001)*. IEEE Computer Society Press, November 2001.

[3] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[5] D. Heuzeroth, T. Holl, and W. Löwe. Combining static and dynamic analyses to detect interaction patterns. In *Proc. of the 6th International Conference on Integrated Design and Process Technology*, June 2002.

[6] R. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-based reverse-engineering of design components. In *Proc. of the 21st International Conference on Software Engineering, Los Angeles, USA*, pages 226–235. IEEE Computer Society Press, May 1999.

[7] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proc. of the 3rd Working Conference on Reverse Engineering (WCRE), Monterey, CA*, pages 208–215. IEEE Computer Society Press, November 1996.

[8] J. Niere, W. Schäfer, J. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348, May 2002.

[9] J. Niere, J. Wadsack, and L. Wendehals. Handling large search space in pattern-based reverse engineering. In *Proc. of the 11th International Workshop on Program Comprehension (IWPC), Portland, USA*, May 2003.

[10] A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, May 1994.

[11] SUN Microsystems. *AWT, the SUN Java Abstract Window Toolkit. Online at http://java.sun.com/products/jdk/awt*.

[12] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proc. of the 9th International Conference on Software Maintenance (ICSM), Oxford, UK.*, pages 230–238. IEEE Computer Society Press, September 1999.

[13] L. Wills. Using attributed flow graph parsing to recognize programs. In *Proc. of International Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 1073, Williamsburg, Virginia, 1994, November 1996. Springer Verlag.

An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls[*]

Abdelwahab Hamou-Lhadj
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
ahamou@site.uottawa.ca

Timothy C. Lethbridge
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
tcl@site.uottawa.ca

Abstract

Examining the behavior of a large legacy software system helps understand its functionality. Dynamic analysis techniques are well suited for this purpose. Run-time information is typically represented in the form of execution traces; however, the amount of information contained in a trace, of even a small program, can be very large and usually overwhelming. It becomes important to filter these traces and present only the information that adds value to the comprehension process. Many researchers agree that analyzing recurrent patterns in a trace can be useful to bridge the gap between low-level system components and high-level domain concepts. This paper introduces an efficient algorithm that extracts patterns of procedure calls of large execution traces. We also present a set of matching criteria that can be used in procedural as well as object oriented software systems to decide when two patterns can be considered equivalent.

Keywords:

Reverse engineering, program comprehension, dynamic analysis, execution traces, trace patterns

1. Introduction

Understanding a poorly documented software system is not an easy task. Program comprehension techniques aim at overcoming this difficulty. Tools based on these techniques can indeed help software maintainers to complete their daily tasks in a more efficient way [9]. In general, reverse engineering tools can be categorized according to whether they perform a static analysis of the code or a dynamic analysis of the executing system. In [10], Stroulia and Systä presented a large set of reverse engineering activities where dynamic analysis can be used, such as, extracting system modularization, understanding the role of software artifacts and so on. Many other researchers use run-time information to solve

the popular problem of feature localization – locating low-level system components that implement a particular software feature [4, 5, 13]. Moreover, Zayour and Lethbridge [14] experimented with a large real world telecommunication system and found that traces of procedure calls, once made usable, can be very useful to help maintainers perform cognitively taxing activities. Their tool, called DynaSee, uses techniques such as redundancy removal, pattern detection and routine ranking to overcome the size explosion problem of run-time information. Among the features of DynaSee is the possibility for software engineers to replace a pattern of procedure calls (called trace pattern) with a textual description mapping low-level system components to high-level application domain concepts. However, they did not present an algorithm that detects these patterns.

In this paper, we present an efficient algorithm that extracts trace patterns. We also present a list of pattern matching criteria that can be used in procedural software systems to group similar but not necessarily identical patterns together. Our algorithm is based on a technique used to solve a problem known as the common subexpression problem [3, 6]**,** which consists of transforming a rooted tree into its most compact form in such a way that all isomorphic subtrees are represented only once. Figure 1. illustrates this concept.
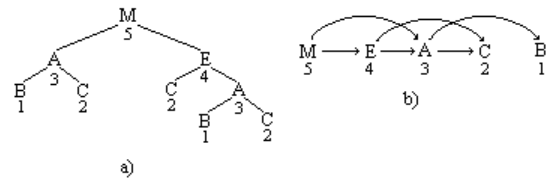

Figure 1. The graph b) represents the compact form of the tree a)

Jerding et al. [8] presented an algorithm that is similar, in principle, to the one provided in this paper. However, their algorithm has some limitations, as we will see in the related work section. The rest of this paper is organized as follows; the next section presents related

---

work. We define what we mean by trace patterns in Section 3. The algorithm that detects them is explained in Section 4. Section 5 describes a set of matching criteria that can be used to decide when two patterns are equivalent. Finally, we conclude in Section 6.

2. Related work

Jerding et al.[8] emphasized the importance of trace patterns for understanding the behavior of object oriented systems. They also presented an algorithm that identifies them. However, their algorithm considers all kinds of repetitions as patterns. This is probably due to the requirements of their visualization tool. For example, they considered contiguous repetitions as trace patterns (that is, candidate high-level concepts) at the same level as non-contiguous repetitions. We think that contiguous redundancies encumber the trace and do not add value to its content. They should be removed and replaced by the number of their occurrences, if necessary. The same choice was made by Zayour and Lethbridge [14] and De Pauw et al. [2]. In addition to that, their algorithm considers identical matches only.

De Pauw et al. [2] considered patterns that are similar but not necessarily identical and presented an interesting list of matching criteria. However, they briefly discussed the algorithm that detects them. In addition to that, most of their matching criteria apply to object oriented systems only.

3. Definition of a trace pattern

Ideally, a trace pattern captures a high-level domain concept. In procedural software systems, these concepts are usually implemented in the form of interactions between the system procedures. Zayour and Lethbridge define a trace pattern as "a sequence of calls that occurs repetitively but non-contiguously in several places in the trace" [14]. This definition excludes patterns that are not identical but that exhibit some similarities. We add to this definition the fact that instances of this sequence of calls do not need to be identical but satisfy some pattern matching criteria. Enabling fuzzy similarity can be very beneficial to trace compression and visualization. The pattern matching criteria can vary depending on the system at hand. They can be either specified by the users or extracted automatically using heuristics.

4. The algorithm

A trace of procedure calls can be represented by a rooted, ordered, labeled tree. Each node corresponds to a procedure call. The node label can be the name of the procedure. The tree levels correspond to the nesting levels of the calls. A trace pattern is then represented as a repeated subtree. Our algorithm starts with a preprocessing stage that aims at removing contiguous

repetitions due to loops and recursion. In [7], we presented a simple but efficient algorithm that does this. The hierarchical nature of the trace is maintained by adding a virtual call whose label starts with *Seq* followed by the number of occurrences of the repeated sequence. Please, note that this virtual call can be omitted in case of repetitions of single procedure calls as illustrated in Figure 2.
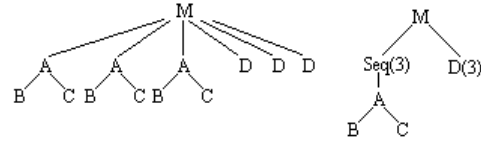

Figure 2. Removing contiguous repetitions

Now that the trace is preprocessed, we apply the pattern detection algorithm to extract trace patterns. As mentioned earlier, the idea behind this algorithm is based on transforming a rooted, ordered, labeled tree to its most compressed form by representing repeated subtrees only once. The result of this compression is a directed acyclic graph as shown in Figure 1. Flajolet et al. described a top-down recursive procedure that solves this problem in an expected linear time assuming that the degree of the tree is bounded by a constant [6]. Valiente presented an iterative version of Flajolet et al.'s algorithm with a slight improvement of its readability [12]. In our previous work, we used an adaptation of Valiente's algorithm to compress a trace of procedure calls [7]. In what follows, we extend it to consider similar but not necessarily identical patterns as well as enabling the frequency analysis of the patterns.

Before getting into the details of the algorithm, first, consider a function called *Match(n1, n2)* that takes two nodes *n1* and *n2* and returns true if the trees rooted at these nodes are considered similar according to predefined matching criteria. The function returns false otherwise. We discuss the specifics of this function in Section 5.

The algorithm proceeds by traversing the tree in a bottom-up fashion (from the leaves to the root). Each node is assigned a certificate (a positive integer between 1 and n, where n represents the size of the tree). The certificates are assigned in such a way that two nodes *n1* and *n2* have the same certificate if and only if *Match(n1, n2)* returns true, that is, the trees rooted at them exhibit some similarities but are not necessarily isomorphic as is the case in Valiente's algorithm.

To compute the certificate, the algorithm uses a signature scheme that identifies each node. The signature of a node *n* consists of its label and the certificates of its direct children, if there are any. A global hash table is used to store the certificates and signatures and ensure that similar subtrees will always hash to the same

element. We added a new field to the table in order to select only patterns that satisfy a certain frequency threshold. Table 1. shows the resulting table that corresponds to applying the algorithm to the tree of Figure 1. The frequency field enables the frequency analysis of the trace. T. Ball showed that frequency analysis of dynamic information can help programmers cluster components according to their behavior and identify related computations [1].

Table 1. Result of the algorithm when applied to the tree of Figure 1.

Certificate	Signature	Frequency
1	B	1
2	C	1
3	A 1 2	2
4	E 2 3	1
5	M 3 4	1

The complexity of the algorithm consists of the time it takes to traverse the tree, the time it takes to compare two subtrees, i.e. compute the function *Match,* and the time it takes to compute the signatures. If exact match is selected and the degree of the tree is bounded by a constant, the algorithm performs in expected linear time.

One can easily see that the resulting table contains a compressed form of the tree. The last step of the algorithm is to walk through the table and extract the patterns that satisfy a given frequency threshold. The table is, first, sorted in order of descending certificates, i.e. the first element of the table is the one that has the highest certificate (this corresponds to the certificate of the root). We use a recursive procedure to display the components of each pattern. The frequency threshold can be specified by the user. Future work should focus on determining it automatically.

5. Pattern matching criteria

De Pauw et al. [2] studied situations where two sequences of calls can be considered as instances of the same pattern in object oriented systems. As a result they presented a list of matching criteria. We found that some of these criteria, namely, identity, repetition, depth-limiting and commutativity can be applied to procedural software systems as well. In this section, we explain these criteria and introduce three new ones: utility, distance and flattening. The design of the function *Match* depends on the selected matching criteria. Some of these criteria can be combined together. Future work should determine how.

5.1 Identity

The identity criterion is probably the simplest one to compute. Two sequences of calls are similar if they have the same topology, which mean, they have the same call structure, order of calls and so on. This criterion might be useful for novices who wish to construct an initial understanding of the trace.

5.2 Repetition

The number of repetitions of contiguous sequences of calls does not really add too much value to the trace. These repetitions can be ignored. For example, the two subtrees of Figure 3 can be considered as instances of the same pattern.
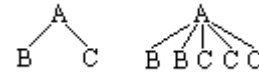
Figure 3. Repeated sequences can be ignored when looking for patterns

5.3 Ordering

This matching criterion is based on the commutative criterion presented in [2] without the restriction of considering objects of the same classes only, since, we do not deal with objects here. If the order of calls does not matter to software engineers, then it can be ignored. To generalize the algorithm to unordered trees, we need to sort the certificates that appear in the signatures before comparing them. If this criterion is used, it will certainly be beneficial to users who already have a certain understanding of the system. Future work should focus on determining the importance of the order of calls according to the tree levels where they occur. For example, the order may not be important at the leaf level where utility procedures are used. This is not necessary the case at higher levels.

5.4 Depth-Limiting

Depth-limiting allows comparing two subtrees up to a certain depth. The calls that are beyond this depth are ignored. In a layered system, components of one layer communicate with the components of the layer below. Patterns of the same layer can be grouped together. This is useful to users familiar with the system architecture. We intend to experiment with different execution traces to determine at which level of the trace tree this criterion could be applied.

5.5 Utility

Utility procedures are domain independent routines that implement specific tasks (e.g. sorting an array). Users may decide to ignore them when comparing patterns. There are different heuristics that are used to detect such procedures (e.g. compute fan-in and fan-out). Consider the two sequences of calls in Figure 4., where u1, u2, u3 and u4 are utility procedures. These two sequences can be considered similar if we decide to ignore the utility procedures.

One way of implementing this concept is to group the utility procedures in one subsystem and then go through the trace and replace their occurrences by the name of this subsystem. This results in a trace with a higher level of abstraction.
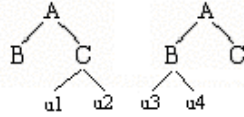
Figure 4. These two sequences can be considered similar if the utility procedures are ignored

5.6 Distance

Two patterns may have almost the same procedure calls but slightly different structures. For example, a control statement can lead to different execution paths depending on the program inputs. That is, the same program behavior might result in slightly different sequences of procedure calls. We would like to be able to group these sequences together as being one common pattern. For this purpose, we need to evaluate the difference between their structures. The tree edit distance can be used [11]. This criterion might be useful to expert users who are already familiar with the source code.

5.7 Flattening

This criterion does not consider the hierarchical structure of the patterns at all. Instead, it flattens them into a linear structure and compares them. If the same calls exist more than once then they are reduced to one occurrence. This subsumes most of the criteria presented in this paper and will certainly result in a very good compression rate. However, we need to analyze situations where it could be applied usefully.

6. Conclusion and future work

Dynamic analysis is important to understand the behavior of any software system whether it is based on OO concepts or not. Dynamic analysis tools should be as important as static analysis tools. In fact, the combination of both provides, without any doubt, the best solution to address program comprehension issues.

Patterns of procedure calls can be used to bridge the gap between low-level system components and high-level domain concepts. In this paper, we showed an algorithm that extracts them in an efficient manner. We also presented a set of matching criteria that can be used, in conjunction with the ones presented in [2], to group similar patterns. Future work should focus on validating these criteria and classify their usage according the user's knowledge of the systems. The long term goal is to

determine heuristics that automatically select patterns that most likely correspond to high-level concepts.

References

[1] T. Ball, "The concept of dynamic analysis", *ACM SIGSOFT Software Engineering Notes*, v.24 n.6, , Nov. 1999, pp.216-234

[2] W. De Pauw, D. Lorenz, J. Vlissides and M. Wegman, "Execution Patterns in Object-Oriented Visualization", In Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98), USENIX, 1998, pp. 219-234

[3] J.P. Downey, R. Sethi and R.E. Tarjan, "Variations on the common subexpression problem", *J. ACM.* 27, 1980, pp. 758-771

[4] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis", *ICSM*, 2001

[5] T. Eisenbarth, R. Koschke, D. Simon, "Feature-Driven Program Understanding Using Concept Analysis of Execution Traces", *IWPC*, 2001

[6] P. Flajolet, P. Sipala, J.–M. Steyaert, "Analytic variations on the common subexpression problem", *In Automata, Languages, and Programming*, Springer-Verlag, 1990

[7] A. Hamou-Lhadj, T. C. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces", *IWPC*, 2002

[8] D.F. Jerding, J.T. Stasko, T. Ball, "Visualizing Interactions in Program Execution", *ICSE*, 1997

[9] M. –A.D. Storey, K. Wong, H.A. Muller, "How Do Program Understanding Tools Affect How Programmers Understand Programs?", *WCRE*, 1997

[10] E. Stroulia, and T, Systä, "Dynamic analysis for reverse engineering and program understanding", *ACM SIGAPP Applied Computing Review*, 2002

[11] K. C. Tai, "The tree-to-tree correction problem", *ACM*, 26(3):422-433, 1979

[12] G. Valiente, "Simple and Efficient Tree Pattern Matching", *Research report*, *Technical University of Catalonia,* E-08034, Barcelona, 2000

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

[14] I. Zayour and T.C. Lethbridge, "A Cognitive and User Centric Based Approach For Reverse Engineering Tool Design", *CASCON*, 2000

Towards Differential Program Analysis

Joel Winstead and David Evans
Department of Computer Science
University of Virginia
{jwinstead,evans}@cs.virginia.edu

Abstract

Differential Program Analysis is the task of analyzing two related programs to determine the behavioral difference between them. One goal is to find an input for which the two programs will produce different outputs, thus illustrating the behavioral difference between the two programs. Because the general problem is undecidable, an unsound or incomplete analysis is necessary. A combination of static and dynamic techniques may be able to produce useful results for typical programs, by conducting a search for differentiating inputs guided by heuristics. This paper defines the problem, describing what would be necessary for this kind of analysis, and presents preliminary results illustrating the potential of this technique.

1. Introduction

Notkin has argued that the future of program analysis lies in analyzing multiple versions of the same program together [4]. This allows us to amortize the cost of analysis across the development cycle, as well as to direct analysis efforts towards differences, and may allow kinds of analysis that would otherwise be intractable. We agree that this is a good strategy, and further argue that analyzing two versions of a program to find a behavioral difference is an important problem not just because it can reduce the cost of analysis, but because finding behavioral differences is a useful goal in itself: it can aid in understanding and maintaining a program as well as in recognizing unintended side effects of modifications.

When making a change to a program, either to correct a known error or to add a new feature, the consequences of the change are not always fully understood. The change may have unintended side effects that were not anticipated by the programmer, or may fail to accomplish the intended goal. The change may even have no effect at all. In order to prevent unintended side effects and verify that changes have the intended effect, it would be helpful to have an automated analysis showing the actual effect of the modification on the program's behavior.

Programs are frequently maintained by people who are far removed from the original development process. The intended purpose of modifications in the program's history is not always clear or documented. The actual effect on the program's behavior of the presence of a particular part of the program may be unknown; a particular line may be crucial or it may have no effect at all. An analysis that shows the difference in behavior caused by the presence or absence of a particular element would assist maintainers in understanding the program.

Testing and dynamic analysis of programs could also benefit from this sort of analysis. When a change to a program is made, it is important that it is well tested. New tests may need to be added to the regression test suite to test the modification adequately. Many dynamic analysis tools depend on the quality of the test suite for a program, and may produce incorrect results if no tests exist that exercise a particular modification.

What is needed is a set of automated techniques to analyze the effect of modifications. We use *differential program analysis* as a general term to describe analyses that focus on the differences between two similar programs. In the sections that follow, we outline what such an analysis needs to do, propose some heuristics and techniques that can be used to do this analysis, and present preliminary results showing the promise of this technique.

2. Problem Definition

One goal of differential program analysis is to generate a test case that demonstrates the difference in behavior between the two programs. We assume that the behavioral difference is small relative to the input space (i.e., the two programs produce identical output for nearly all inputs). While it would be interesting to analyze changes that affect the result of every input to the program, this would require a different kind of analysis. Our goal is to find behavioral differences, not to analyze known ones. Once a difference is

found, existing techniques such as Zeller's Delta Debugging method [9] can be used to analyze the difference.

We concentrate on analyzing two versions of the same program. The structural difference between the two programs must be small relative to the size of the program: only a few lines of code or a few procedures in the program should be different. We would like to develop techniques that take advantage of the similarities between the two programs, rather than use existing techniques to analyze the programs independently and compare the results.

Because our goals include finding unanticipated side-effects of changes, we cannot assume that an existing regression test suite is able to find all interesting behavioral differences. Regression testing finds differences in behavior that were anticipated by the designers (or testers) and specifically checked. While regression test selection [1] is a useful technique for reducing the cost of testing, it cannot reveal new differences that are not already tested by the suite. We also would like to be able to analyze undocumented programs that may not have test suites.

We assume we have a generator capable of producing a differentiating test case, but that it is not reasonable to do an exhaustive search of the input space. It is not necessary for all generated inputs to be valid; the search will eliminate inputs that both programs consider to be errors. If the difference in behavior is small relative to the input space, and we have a generator that can produce the right inputs, the analysis problem becomes one of performing a directed search to find inputs which reveal behavioral differences.

3. Approach

This kind of analysis requires solving several subproblems: we must find inputs that reach the syntactic difference, generate differences in state between the two programs, and propagate these differences to the output.

The two programs will always produce the same output for a given input unless, at some point, they execute different instructions. Therefore, in order to find test cases that result in different output, we must first figure out how to reach the syntactic changes to the program. This subproblem is itself undecidable, but incomplete solutions have been proposed for it using search techniques such as simulated annealing [6] or genetic algorithms [3] [5]; these techniques use fitness functions to generate test inputs that reach particular parts of a program. Symbolic execution and constraint solving is also a possible approach.

Once the syntactic changes in the program have been reached, it is also necessary for a difference in state to result, and for this difference to be propagated through the programs far enough to result in different output. It is possible for a modification to produce changes in intermediate values without producing any difference in the end result.

Narrowing the input space to inputs that reach the modification will not always be sufficient: it will still be necessary to search this space to find inputs that result in actual differences in output.

Tracey et al. [6] show how to use simulated annealing to evolve test inputs that cause a program to reach a specified point in the code. This is accomplished by determining what branches the program must take to reach the point of interest, and developing a fitness function that evaluates how close the program is to taking the correct branches. For branches that the program should take, the condition for the branch is transformed into an expression that measures how close the program is to taking the branch, and these expressions are combined to form a fitness function which is used to evolve test inputs that cause the program to reach the desired point. In order to apply this technique to finding differences between programs, new fitness functions must be constructed that compare the behavior of the two programs and guide the search towards input that is likely to reveal differences.

In order to direct the search towards an input that produces an actual behavioral difference, we must have some way to measure how close a particular input is to achieving this goal even if the goal has not yet been reached. While not all inputs will produce a difference in output, some inputs may produce different intermediate values, or other measurable differences in execution that may be important cues for finding inputs that produce a behavioral difference. We propose several heuristics that may be useful for guiding a search towards inputs that produce actual differences in output.

First, we note that boundary conditions in the two programs indicate what decisions the programs are making, and that differences in behavior often lie along boundary conditions. Selecting test cases that exercise boundary conditions in the two programs is a promising way to find differences. Focusing attention on boundaries that exist in one program but not the other is particularly interesting, because these decisions lead to paths that are not in both programs, and reaching these paths may reveal different behavior. This last heuristic takes advantage of known similarities between the programs to focus on the difference, and has the potential to be more useful than approaches that analyze the programs separately.

If we evolve test sets, instead of evolving single test cases, there are additional heuristics we can use. We can select for test sets that maximize the total number of paths executed, or other coverage metrics. We could modify these coverage metrics to include only those paths that reach the syntactic difference between the programs, which also takes advantage of the similarities between the programs.

Evolving test sets also allows us to compare the ways the two programs map the input space into paths through

the program. If program P_1 maps two inputs I_1 and I_2 to the same path, but program P_2 maps inputs I_1 and I_2 to two different paths, this reveals something about how the programs divide the input space, even if the output is the same. Selecting for test sets that do not produce isomorphic mappings from inputs to paths in this way may lead to revealing behavioral differences because they indicate regions of the input space where the two programs do not handle the input in the same way.

In addition to these heuristics, ongoing work by Xie and Notkin [8] examines comparing program spectra combined with various heuristics to identify possible faults in modified programs even in cases where no actual differences in output result. Program spectra are signatures of program behavior (such as the distribution of paths taken, procedures executed, and values modified by the program) that can be used to characterize the program's execution. Harrold et al. [2] also investigate the effectiveness of various program spectra in identifying differences between programs. These studies focus on identifying differences in execution that occur in regression tests of a program, not on guiding a search for inputs that produce actual behavioral differences. However, some of the heuristics identified there may be useful in constructing fitness functions that are useful for this purpose.

If no differentiating test case can be found, this does not necessarily mean that no behavioral difference exists. It would be useful to have some way of measuring how thorough a search has been conducted, because this would provide a measure of confidence that the two programs actually have the same behavior. This could be done by estimating how much of the relevant search space has been tested, or by coverage of the modified parts of the programs. Mutation analysis [7] has been used to evaluate the effectiveness of a test suite by how well it can identify differences between the original and modified programs. It may be necessary to develop new coverage metrics that take into account the special problem of covering differences in code.

4. Preliminary Results

We have developed a small system to explore some of the techniques described above. The system uses less than 1,000 lines of Java code, and is capable of evolving test cases that show behavioral differences in small programs. The programs must be instrumented by hand to compute the fitness functions, but this could be automated in the future. We will illustrate the system using a simple example.

The short procedure in Figure 1 classifies a triangle by comparing the lengths of its three sides: it returns a value indicating whether or not the three lengths given can form a triangle, and if so, whether the triangle is equilateral, isosceles, or scalene, and whether the largest angle is right, obtuse,

```
int classify( int a, int b, int c) {
    int kind = UNKNOWN;
    if (a + b <= c || b + c <= a || c + a <= b)
        return INVALID_TRIANGLE;

    if ( a*a + b*b == c*c || b*b + c*c == a*a
        || c*c + a*a == b*b)
        kind |= RIGHT_TRIANGLE;
    else if ( a*a + b*b > c*c
            && a*b + c*c > a*a
            && c*c + a*a > b*b)
        kind |= ACUTE_TRIANGLE;
    else
        kind |= OBTUSE_TRIANGLE;

    if (a==b || b==c || c==a )
        if (a==b && b==c)
            kind |= EQUILATERAL_TRIANGLE;
        else
            kind |= ISOSCELES_TRIANGLE;
    else
        kind |= SCALENE_TRIANGLE;

    return kind;
}
```
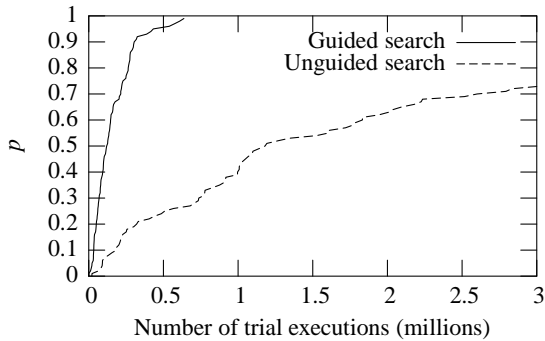
Figure 1. Triangle classification procedure

or acute. A different version of this procedure (not shown) lacks the || c==a (shown in the box).

This procedure is simple enough that we can easily see the effect of the modification by inspecting it: it will, in some cases, incorrectly classify an isosceles triangle as scalene (for example, the triangle with sides (3,4,3) would be classified as scalene, even though sides a and c are the same length). However, we will use it as an example to demonstrate how an automated tool could generate test cases that demonstrate the difference using the boundary condition heuristic.

The idea behind the heuristic is that test executions that are at or near boundary conditions in the program are more likely to reveal differences. We can develop such test cases by instrumenting the program to compute a measure of nearness to boundaries, and use this measure to guide the search for differentiating test cases. At each decision point in the program, the conditional expression is converted into an expression measuring how far the program was from making the opposite decision, similar to the technique Tracey et al. used to select inputs that reach a particular point in the program [6]. We use the minimum of all of these values to measure how close the program was to taking a different path for a particular test case. We can then use this fitness function to guide a genetic algorithm to

Figure 2. Probability of finding difference vs. number of trial executions

evolve test cases that cause the program to reach boundary conditions.

Preliminary results show that this technique works for the triangle classification example given above: the guided search was able to find a differentiating test case using significantly fewer test executions than a random, unguided search. The graph in Figure 2 shows that fifty percent of the time, the guided search was able to identify the difference in 120,000 trial executions or fewer, while the unguided search required over a million trial executions before having a 50% probability of finding one.

This technique works particularly well for this (admittedly contrived) example because the behavioral difference lies along one of the boundary conditions: c==a. For programs that have more complicated control flow, and a more complicated relationship between the input and the control flow, this is not sufficient. However, there are several ways this technique can be improved. We are currently exploring focusing on decisions that are made in only one program but not in the other in order to guide the search, rather than looking at all boundary conditions. We are also examining the use of static analysis to determine which decisions are most important and which decisions must be made to reach the changed portion of the program.

5. Summary

The modern development process, using version control systems like CVS and online repositories such as Source-Forge, makes available many related versions of the same programs. We should take advantage of this opportunity to analyze related versions of programs to better understand them. It is important to develop techniques to analyze two versions of the same program together, not only because it could reduce the overall cost of testing and analysis, but because it could reveal important facts about the differences

between the programs. This kind of information would be useful in avoiding unintended side effects, understanding the development history of undocumented programs, and in identifying the actual effects of particular parts of the program.

Many of the hard problems that must be solved before we can achieve the goals of differential program analysis have undergone rapid progress recently, such as using global search techniques to evolve inputs that reach particular parts of programs [6], and using program spectra to compare related versions of the the same program [2] [8]. We are optimistic that we are near the point when techniques can be combined in ways that enable useful and revealing analyses of the differences between two similar programs.

References

[1] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, 2001.

[2] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[3] C. C. Michael, G. E. McGraw, M. A. Schatz, and C. C. Walton. Genetic algorithms for dynamic test data generation. Technical Report RSTR-003-97-11, RST Corporation, Sterling, VA, May 1997.

[4] D. Notkin. Keynote: Longitudinal program analysis. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE 2002)*, page 1, Charleston, SC, November 2002.

[5] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification, and Reliability*, 9(4):263–282, 1999.

[6] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *Internation Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180. IFIP, January 1998.

[7] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.

[8] T. Xie and D. Notkin. Checking inside the black box: Regression fault exposure and localization based on value spectra differences. FSE Poster Session, November 2002.

[9] A. Zeller and R. Hildebranst. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

Languages for Dynamic Instrumentation

Steven P. Reiss, Manos Renieris
Department of Computer Science
Brown University
Providence, RI 02912-1910
401-863-7641, FAX: 401-863-7657
{spr,er}@cs.brown.edu

Abstract

Dynamic instrumentation has proven to be a valuable technique for a variety of program analyses. However, developing a new analysis based on dynamic instrumentation is difficult, error prone, and time-consuming. One solution is to develop a common framework that would enable quick and easy dynamic instrumentation for a variety of applications. Developing a practical solution along these lines, however, requires that we understand and effectively model how instrumentation can and should be used. We suggest that an event-oriented framework based on program analysis might be a viable approach to achieving such a practical solution.

1. Motivation

Dynamic analysis has been used for a wide variety of different applications, from simple profiling to program understanding. We have been using it in a variety of projects for performance analysis, visualization, program modeling, and fault location. In most of the applications of dynamic analysis, the technique has demonstrated itself as an invaluable tool that is able to provide insights far beyond those of static analysis.

Even so, dynamic analysis still sees only limited use in day-to-day applications, in today's programming environments, and by most programmers. There are several reasons for this disparity, but most rise from the fact that dynamic analysis is expensive, both in terms of the overhead involved in collecting the appropriate data, in terms of developing practical instrumenters, and in terms of developing tools that can use the data.

What is needed is a framework to support dynamic analysis that could be used practically for a variety of different applications. If such a framework existed, it would be relatively easy to develop new applications of dynamic analysis and to incorporate them into today's programming tools. Some requirements, like minimizing instrumentation overhead, are often too difficult to achieve for any single application. Developing a framework that addresses them would empower developers to use dynamic-analysis based tools as part of their everyday programming.

In this position paper, we outline some of the interesting research issues that arise in attempting to define (and later implement) such a framework.

2. Requirements

A practical dynamic analysis framework has to meet a broad range of requirements. These are needed both to make it applicable to a variety of different applications and to ensure that it can be used on a variety of real systems. These requirements include:

- Low usage overhead. The user should have to do as little as possible to get the leverage of the tool. At best, the user could run the tool externally, as with valgrind [7]. At worst, the tool should require recompilation, but in this case it should be integrated with the compiler, and provided as an option within it or a script around it.

- Low execution overhead. The resource requirements of the tool should be minimal. Minimizing the tools' overhead by intelligent, problem-specific instrumentation should be the primary goal of the framework.

- Static selectivity. The user should be able to specify what portions of the system should be instrumented and what data should be collected. This should be available at as fine a level as possible. The selection of what to instrument and what data to collect should be based on the structure and semantics of the program. This implies that dynamic analysis should be predicated on some underlying static analysis.

- Source anchoring. The debugging information preserved in binary formats is often inadequate to produce meaningful messages for more complex program analyses, especially in the presence of optimization. Depending on how close the tool is to the compiler, this is more or less of a challenge. From the user perspective, though, it should always seems as if the tool has all the information the compiler has.

- Temporal selectivity. Instrumentation should be limited not only by specific portions of the program, but also by those parts of the execution that are relevant to the task. This temporal information might be determined a priori or dynamically.

- Handling of real programs. A problem with today's dynamic instrumentation tools is they often are not capable of handling the wide range of programs that developers are interested in. The next three requirements follow from this.

- Handling of libraries. Much of the work in today's applications is done inside system or user libraries. To do appropriate analysis, one often needs dynamic information from these libraries. Moreover, to understand the semantics of the application, one must often understand the semantics of the libraries. This requirement becomes more complex when one realizes that source is often not available for many libraries. In any case, an instrumentation framework should at least provide for data collection at the boundary of libraries.

- Handling of multithreaded programs. Java and C# programs are often multithreaded. An instrumenter needs to be able to deal with the underlying complexities both in terms of collecting appropriate data and in terms of not imposing additional synchronization points on the application and thus changing its behavior.

- Handling of whole systems. Many of today's programs are actually multiple-process distributed systems. The analysis and hence instrumentation that needs to be done on these systems will require correlating data accumulated from the different processes into a single analysis.

- Usable Results. Another key problem in today's instrumenters is that the data that is produced is often very specific to a particular application and not easy to reuse in other applications. What we need is a relatively standard data format that can serve as the basis both for immediate and deferred analysis.

Meeting these requirements will be difficult. However, by using the collective experience from current instrumenters, static analyzers, aspect-oriented programming, and other areas, it should be possible to develop an appropriate framework.

3. Framework Overview

We envision a framework that is built on two languages. The first is used to let a tool define what portions of a system should be instrumented and what information is required from those portions. This will be used by a instru-mentation tool to produce one or more event streams describing appropriate portions of the execution.

The second language will let a tool define how these event streams should be processed to produce the data needed for analysis. This could involve generating higher-level events streams, accumulating information, tracking program or object states, or other analysis techniques. The framework would use this description to process the events as they were generated as efficiently as possible.

Central to this framework is the notion that both languages can make direct use of information about the system being analyzed. This means that they should be able to refer to basic blocks, to the definitions and uses of particular variables or fields, to def-use chains, and to particular packages, libraries, and routines.

4. Instrumentation Definition Language

The first part of this framework is dependent on a language that lets the developer describe the information that needs to be collected from dynamic instrumentation at a fine level of detail. The actual instrumentation is addressed by systems like EEL [4], SOOT [9], or JikesBT [3]. We after a language similar to the languages for specifying pointcuts in aspect-oriented programming [6,8].

This language should be geared toward generating event streams. Events are a general purpose mechanism that closely matches the methodology of run time instrumentation. The underlying framework will have to deal with many types of parameterized events, including:

- Call/Return of a method;
- Definition/Use of a value;
- Enter/Exit of a basic block;
- Throw/Catch of an exception;
- Create/Start/Stop/Wait/IO/Run of a thread;
- Read/Write of a location or field;
- Allocate/Free of an object;
- Send/Receive of a message;
- Program specific events.

The set of events that are relevant to a particular program or run needs to be specified in a high level way. This will sometimes be done globally (e.g. interest in all call/return events for profiling), and sometimes very program specific (e.g. when does field X change in method Y; when is method A called with parameter B). Moreover, the set of events generally should be independent of the code.

In both cases static analysis of the program, typically done at the byte or machine code level, will be appropriate. This analysis should let one specify, for example:

- That one wants to detect the start of each basic block. The resultant instrumentation could then make use of control flow analysis to minimize the amount and size of instrumentations.

- That one wants to track field accesses for a particular set of field writes. This would require data flow analysis to determine which reads in the program might be relevant to the particular writes.

- That one wants to detect calls to a particular set of methods for objects allocated at a certain point in the program. For example, one might want to check that a particular instance of a Java iterator is used correctly.

- That one wants to detect reads and writes of shared storage. This would require static analysis to determine what fields can be accessed by multiple threads and which accesses to those fields should be considered shared.

The research in this area is to attempt to put together a language that allows an analysis application to specify what set of events it wants from the program. This could either be a language per se, an XML file describing the set of events, or event an appropriate set of function calls and callbacks.

This language will have to deal with all the issues outlined above — handling a wide range of events, being able to specify those events to apply to the whole program or large portions of it, being able to restrict those events to particular locations based on semantic properties of the program, and allowing a variety of different parameters to be associated with each event.

To leverage such a language it is necessary to build an appropriate implementation. This is again a research problem involving what and how to do the static analysis needed to minimize instrumentation, techniques for dynamically inserting and removing instrumentation, and automatic optimization of instrumentation based on semantic information.

5. Analysis Language

While event streams are a logical conceptual output from an instrumentation front end, what is often needed is the result of analysis based on the event stream rather than the event stream itself. There are several different types of such analysis that are particular to the applications of runtime instrumentation. The inspiration comes from languages for higher level debugging, like COCA [2] and QBD [5].

For visualization and some program understanding applications, it will be desirable to map the event stream into a sequence of higher-level events. This can occur within an event stream (for example, mapping basic block event to program path events), or it might occur among multiple event streams (for example, taking information about monitor entry and exit events from multiple threads and using this to generate events denoting what threads are blocking on what other threads).

For performance analysis and related applications, it is desirable to accumulate information from the event sequence. One might want to look at the total number of calls of each method, the number of allocations of each class, the time spent in each method, or the number of calls of each method pair. This information might be further confined by accumulating information by class or package or event according to higher-level events such as user interface interactions or remote procedure calls.

Another application area for run time instrumentation is involves the dynamic checking of semantic properties of a system. These properties are typically specified using finite state systems (either using pure or extended FSMs, using regular or path expressions, or using a language such as LTL or CTL [1]). What one wants to get out of instrumentation here is whether the actual program run satisfied or did not satisfy the specification. This implies that the sequence of run time events generated by the front end needs to be filtered and then use to check against the underlying automata.

In each of these cases, the appropriate analysis can be done either after the fact or while doing tracing. After the fact analysis is easier in that one can isolate the analysis from the instrumentation and can easily do several different analyses of the same instrumented run. This is advantageous, for example, in software visualization where the user will want to see different views of the run and the exact nature of those views might not be known in advance.

In most applications, however, the raw event streams are going to be substantially larger and more complex than the results of the analysis. Here, it is much more effective to do the event analysis on the fly, storing only the accumulated result. An ideal instrumentation environment should provide a stream-based processing language that would facilitate this. Again, this could be a real language, a high-level XML description of what needs to be done, or simply a reasonable programming interface that facilitates the appropriate processing.

We note that this language and facility will probably need to have access to the semantic analysis that was used in doing the actual instrumentation in order to correctly interpret the events. This information will either have to be recomputed or will be stored in auxiliary files as part of the instrumentation process.

The interesting research issues here are first attempting to determine the appropriate range of analyses that should

be doable dynamically, in determining what is an appropriate interface for doing these analyses, and in providing a very efficient but generic implementation mechanism that will support the analyses. Other research issues that come up involve ways of combining multiple event streams in the analysis milieu and doing all this without significantly affecting the behavior of the program being instrumented.

6. Example Approaches

While we have not built anything that meets the needs outlined above, we have and continue to work on a variety of different approaches that make us believe that the general mechanisms described here can be achieved.

We currently support several different instrumenters for different applications. For software visualization, we have two instrumenters, one for C/C++ and one for Java. Both are capable of instrumenting the user's application and all the appropriate libraries. Both handle multiple threads and offer a limited degree of selectivity as to what information to obtain. While the initial data is obtained as a set of independent event streams, one per thread, this data is processed dynamically into a common sequence. Additional, on-the fly or after-the-fact processing can be done within the system for a wide variety of different resultant analyses. Still, one of the best experiences we have had with such systems was in minimally modifying the profiling library of the compiler to produce a trace of function calls and returns. The system was very easy to implement. It was also very easy to use, since it required as much effort as profiling.

For dynamic visualization of software, we have developed an instrumenter that accumulates a variety of data over millisecond time intervals and passes the accumulated data to a front end. Using a variety of techniques, we were able to limit the performance loss due to instrumentation (which includes every call, return, allocation, thread state change, and synchronization event) to a factor 2-3.

Finally, we are developing a tool for checking finite state properties of programs through a combination of static and dynamic checking. Given a description of a program property, this tool is able to find the relevant locations in the source that affect that property, determine whether the property needs to be checked dynamically or if it can be determined statically, and, in the case where dynamic checking is necessary, it actually determines exactly what instrumentation is needed to check the property.

7. Acknowledgements

8. References

[1] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. Model Checking. The MIT Press, Cambridge, Massachusetts, 1999.

[2] Mireille Ducasse. Coca: A debugger for C based on fine grained control flow and data events. In Proceedings of the 21st International Conference on Software Engineering, pages 504-515. ACM Press, May 1999.

[3] Chris Laffra, Doug Lorch, Dave Streeter, Frank Tip, and John Field. Jikes bytecode toolkit. http://www.alphaworks.ibm.com/tech/jikesbt.

[4] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In Proceedings of the ACM SIGPLAN'95 Conference on Program- ming Language Design and Implementation (PLDI), pages 291-300, La Jolla, California, 18-21 June 1995.

[5] Raimondas Lencevicius, Urs Holzle, and Ambuj K. Singh. Query-based de- bugging of object-oriented programs. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Ap- plications (OOPSLA-97), pages 304-317, October 1997.

[6] Hidehiko Masuhara and Gregor Kiczales. Modular crosscutting in aspect- oriented mechanisms. In Proceedings of the 2003 European Conference on Object-Oriented Programming, 2003.

[7] Julian Seward. Valgrind. http://developer.kde.org/~sewardj.

[8] David B. Tucker and Shriram Krishnamurthi. Point-cuts and advice in higher-order languages. In Proceedings of the 2003 International Conference on Aspect-Oriented Software Development, Boston, Massachusetts, March 2003.

[9] Raja Vall'ee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java Optimization Framework. In Pro- ceedings of CASCON 1999, pages 125-135, 1999.

Some Axioms and Issues in the UFO Dynamic Analysis Framework

Clinton Jeffery
Department of Computer Science
New Mexico State University
jeffery@cs.nmsu.edu

Mikhail Auguston
Department of Computer Science
Naval Postgraduate School
maugusto@nps.navy.mil

Abstract

UFO is a framework for constructing dynamic analysis tools that require varying degrees of access and control over program executions. UFO combines run time and post-mortem techniques to perform required analyses. Declarative and imperative notations are provided for constructing monitors at appropriate semantic levels. Multiple analyses can be bundled into a given monitor, and multiple monitors can be applied to a given target program execution. This paper presents the central tenets of UFO, along with our current set of research challenges.

1. Motivation

Automatic debugging and program visualization are two of the most promising application areas of dynamic analysis, with potential to impact on crucial areas of software development and maintenance. We believe the slow rate of advancement in these areas is due to the high cost of developing new tools. We have previously focused on a language (FORMAN) and an architecture (Alamo) that reduce these costs [1][2][4]. FORMAN is a special-purpose language for expressing dynamic analyses; it has been implemented previously for subsets of Pascal and C. Alamo is a lightweight architecture for program execution monitoring; it has been implemented for a subset of C and for the virtual machine used by the Icon and Unicon programming languages. The virtual machine implementation of Alamo is attractive for research because it provides high performance and superior ease of use for a full-size "real" programming language, allowing testing on large programs and the possibility of deploying successful tools to a user community.

We recently merged the FORMAN and Alamo efforts to produce UFO (Unicon-FORMAN), a framework for rapidly constructing dynamic analyzers [3][4]. We have used UFO to construct a variety of simple automatic debuggers and visualization tools that run well on small and medium sized applications. Our next efforts must walk the tightrope of scaling up to production tools for large applications, while retaining the power and ease of use that are characteristic of the current research UFO system. With that in mind, this paper presents the central tenets of the UFO system, and concludes with an exploration of the current research problems and our plans to address them.

2. Axioms

UFO is primarily an implementation of FORMAN built on top of the Alamo monitor architecture. Early experiments showed the marriage to improve FORMAN speed by two orders of magnitude and shorten the lines of code necessary to write Alamo monitors by one order of magnitude. This section sketches the primary characteristics of UFO.

- A precise program behavior model, in which semantics of the monitored language are mapped to directed acyclic graphs of events. These graphs are defined using an *event grammar*, a notation that approximates the semantics of the language to be monitored. The behavior model is essential to provide general purpose capabilities for a wide range of tools.
- A declarative special-purpose monitoring language, tailored specifically for dynamic analyses expressed in terms of patterns within the graphs of events. This component is necessary to reduce the cost of developing new tools. Section 4 provides some examples; shorthand refinements to improve the syntax could be explored after the main semantics and performance issues are resolved.
- An hybrid execution model, in which most analysis work is performed at run-time, and more complex analyses transparently combine run-time collection and partial analysis with more extensive post-mortem analysis. This element is necessary but not sufficient by itself to achieve acceptably high performance for large scale production systems. This important element is new in UFO, compared with previous FORMAN and Alamo efforts. It provides high performance.
- Automatic instrumentation provided by special-purpose virtual machine support; static or dynamic configuration of VM instrumentation; no recompilation, relinking, or alteration of target program executables to be monitored. This provides substantial ease of use.

3. Some Research Issues and Challenges

UFO's chief design goals revolve around notational power and ease of use. The current prototype implementation of UFO [5][5] processes millions of events per minute. But, for large programs higher performance is needed. This goal motivates several open problems we are pursuing.

Minimizing the number of context switches. UFO's run-time execution model is based on lightweight coroutine switches between monitors and the program being observed. This separation is a compromise between intrusive in-line single-thread execution used in low-cost analysis tools such as profilers, and the complete separation imposed by high-cost analysis tools such as debuggers. One research goal is to retain the abstraction and low-intrusion benefits of the coroutine model without having to pay (so much) for it.

Virtual machine configuration and customization. The VM instrumentation can be turned off at multiple levels, including compile-time via #ifdef and run-time via a dynamic filter that controls whether instrumented or uninstrumented versions of functions are called, and whether an event report (via lightweight context switch) is performed for a given instrumentation site. This configuration can be further exploited by having the UFO compiler generate a custom VM with exactly the instrumentation it needs for a particular monitoring application. The central VM interpreter function (interp()) can benefit from a finer granularity of customization than the current instrumented-versus-uninstrumented options; it is critical to performance and contains 30 of the 119 types of events instrumented in the VM. Generating a custom VM may greatly improve monitoring performance within this VM interpreter loop. The VM generation system needs to make it easy and convenient for the UFO compiler to generate custom VM's and associate them with generated analyzers in a persistent manner. Custom VM's should be shareable by monitors that use the same events.

Inter-monitor optimizations. When multiple analyses are compiled together, substantial cost savings might be obtained by factoring common tasks such as event data collection. For example, a profiler that computes summaries and a visualizer that shows run-time details might operate on the same information, and might even share some common analysis structures.

Meta-events and analysis hierarchies. UFO's event model composes higher level events from lower level ones, but analysis tools create additional information which may constitute the input for higher level analyses. This facilitates the sharing of analysis information among tools, reducing the cost of running multiple tools.

4. Examples of debugging rules

Alamo's goal was to reduce the difficulty of writing execution monitors to be just as easy as writing other types of application programs. UFO supports FORMAN's more ambitious goal of reducing the difficulty of writing automatic debuggers to the task of specifying generic assertions about program behavior.

This section presents formalizations of typical debugging rules. UFO supports traditional precondition checking, or print statement insertion, without any modification of the target program source code. This is especially valuable when the precondition check or print statement is needed in many locations scattered throughout the code.

Example #1: Tracing. Probably the most common debugging method is to insert output statements to generate trace files, log files, and so forth. It is possible to request evaluation of arbitrary Unicon expressions at the beginning or at the end of events. The virtual machine evaluates these expressions at the indicated time moments.

```
FOREACH   A: func_call &
        A.func_name == "my_func"
     FROM prog_ex
  A.value_at_begin(
     write("entering my_func, value of X is:", X) ) AND
  A.value_at_end(
   write("leaving my_func, value of X is:", X) )
```

This debugging rule causes calls to write() to be evaluated at selected points at run time, just before and after each occurrence of event A.

Example #2: Profiling. A myriad of tools are based on a premise of accumulating the number of times a behavior occurs, or the amount of time spent in a particular activity or section of code. The following debugging rule illustrates such computations over the event trace.

```
SAY( "Total number of read() statements: "
     CARD[ r: input & r.filename == "xx.in"
          FROM prog_ex ]
      "Elapsed time for read operations is: "
     SUM [ r: input & r.filename == "xx.in"
           FROM prog_ex   APPLY r.duration] )
```

Example #3: Pre- and Post- Conditions. Typical use of assertions includes checking pre- and post-conditions of function calls.

```
FOREACH A:func_call & A.func_name=="sqrt"
     FROM prog_ex
  A.paramlist[1] >=0 AND
  abs(A.value*A.value-A.paramlist[1]) < epsilon
WHEN FAILS SAY("bad sqrt(" A.paramlist[1
                    ") yields " A.value)
```

4.1   Generic Bug Descriptions

Another prospect is the development of a suite of generic automated debugging tools that can be used on any Unicon program. UFO provides a level of abstraction sufficient for specifying typical bugs and debugging rules.

Example #4: Detecting Use of Un-initialized Variables. Reading an un-initialized variable is permissible in Unicon, but often leads to errors. In this debugging rule all variables in the target program are checked to ensure that they are initialized before they are used.

```
FOREACH V: variable  FROM prog_ex
        FIND D: lhp FROM V.prev_path
                D.source_text == V.source_text
        WHEN FAILS SAY( " uninitialized variable "
                    V.source_text)
```

Example #5: Empty Pops. Removing an element from an empty list is typical of expressions that fail silently in Unicon. While this can be convenient, it can also be a source of difficult to detect logic errors. This assertion assures that items are not removed from empty lists.

```
FOREACH   a: func_call &
        a.func_name == "pop" AND
        a.value_at_begin( *a.paramlist[1] == 0)
     SAY("Popping from empty list at event " a)
```

5.   Implementation Issues

The most important of these issues is the translation model by which FORMAN assertions are compiled down to Unicon Alamo monitors. Debugging activities are written as if they have the complete post-mortem event trace, the DAG with events, event attributes, and precedence and containment relations, available for processing. This generality is extremely powerful; however, for most practical uses we have seen, assertions can be compiled down into monitors that execute entirely at runtime. Runtime monitoring saves enormously on memory and I/O requirements and is the key to practical implementation. For those assertions that require post-

mortem analysis, the UFO runtime system computes a projection of the execution DAG necessary to perform the analysis.

The UFO compiler generates Alamo Unicon monitors from FORMAN rules. Each FORMAN statement is translated into a combination of initialization, run-time, and post-mortem code. Monitors are executed as coroutines with the Unicon target program.

Monitors generated by the UFO compiler reduce complex assertions to the single event loop. Keeping event detection in a single loop allows uniform processing of multiple event types used by multiple monitors. The code generated by the UFO compiler integrates event detection, attribute collection, and aggregate operation accumulation in the main event loop.

Assertions in UFO may use nested quantifiers implying two nested loops, so code generation addresses this issue by flattening the main loop structure, and postponing assertion processing until required information is available. An hybrid code generation strategy performs runtime processing whenever possible, delaying analyses until post-mortem time when necessary. Different assertions require different degrees of trace projection storage; code responsible for trace projection collection is also arranged within the main loop. The following generation template gives a flavor of the UFO trace projection mechanism.

Rules with two nested quantifiers of the form

```
Quantifier A: Pattern_A
     Quantifier B: Pattern_B FROM A
            Body
```

utilize a monitor whose main loop follows the pattern:

```
Main Loop
     Maintain stack of nested A events
     Accumulate events B in a B-list
     If end of event A
       Loop over B-list
            Do Body
     Endif
     If stack of A is empty
            Destroy B-list
End of Main Loop
```

This requires accumulation of a trace projection for B-events and may cause a mild overhead at the run time.

5.1   Optimization Issues

The UFO approach combines an optimizing compiler for monitoring code with efficient run-time event detection and reporting. Since we know at compile time

all necessary event types and attributes required for a given UFO rule, the generated Unicon monitor can be very selective about the behavior that it observes.

For certain kinds of UFO constructs, such as nested quantifiers, the monitor must accumulate a sizable projection of the complete event trace and postpone corresponding computations until all required information is available. The presence of the previous_path and following_path attributes in UFO rules triggers this kind of optimization; previous_path and following_path are used in rules which specify preceding or following contexts for events of interest.

For further optimization, especially in the case of programs containing a significant number of modules, the following FORMAN construct limits event processing to events generated within the bodies of functions F1, F2, … , Fn.

```
WITHIN F1, F2, … , Fn DO
    Rules
END_WITHIN
```

This provides for monitoring only selected segments of the event trace.

Unicon expressions included in the value_at_begin and value_at_end attributes are evaluated at run time.

Some other optimizations implemented in this version are:

- only attributes explicitly used in the UFO rule are collected in the generated monitor;
- an efficient mechanism for event trace projection management, which disposes from the stored trace projection those events that are no longer used after a certain rule has been fully evaluated;
- both event types and context conditions are used to filter events for the processing.

UFO's goal of practical application to real-sized programs has motivated several improvements to the already carefully-tuned Alamo instrumentation of the Unicon virtual machine. We are working on additional optimizations.

We expect that the most promising optimizations are within the generation of instances of Virtual Machine tailored for a particular monitoring task.

6. Conclusions

The architecture employed in UFO could be adapted for a broad class of languages such as those supported by the Java VM or the .net VM. Our approach to dynamic analysis uniformly represents many types of debugging-related activities as computations over traces, including assertion checking, profiling and performance measurements, and the detection of typical errors. We have integrated event trace computations into a monitoring architecture based on a virtual machine.

Preliminary experiments demonstrate that this architecture is scalable to real-world programs.

One of our next steps is to build a repository of formalized knowledge about typical bugs in the form of UFO rules, and gather experience by applying this collection of assertions to additional real-world applications. There remain many optimizations that can improve the monitor code generated by the UFO compiler; for example, merging common code used by multiple assertions in a single monitor, and generating specialized VMs adjusted to the generated monitor.

Acknowledgements

References

[1] M. Auguston, Program Behavior Model Based on Event Grammar and its Application for Debugging Automation, in the Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging, AADEBUG'95, Saint-Malo, France, May 22-24, 1995, pp. 277-291.

[2] Clinton L. Jeffery, Program Monitoring and Visualization: an Exploratory Approach. Springer, New York, 1999.

[3] M. Auguston, A. Gates, M. Lujan, "Defining a Program Behavior Model for Dynamic Analyzers", in the Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97, Madrid, Spain, June 1997, pp. 257-262.

[4] M. Auguston, "Lightweight semantics models for program testing and debugging automation", in Proceedings of the 7th Monterey Workshop on "Modeling Software System Structures in a Fast Moving Scenario", Santa Margherita Ligure, Italy, June 13-16, 2000, pp.23-31.

[5] M. Auguston, C. Jeffery, and S. Underwood. "A Framework for Automatic Debugging", IEEE 17th Intl. Conf. on Automated Software Engineering, Edinburgh, September 2002, IEEE Computer Society Press, pp.217-222

[6] C. Jeffery and M. Auguston. "Towards Fully Automatic Execution Monitoring". Monterey Workshop 2002, Venice, October 2002, sponsored by US Army Research Office and NSF, pp.232-243

[7] Clinton Jeffery, Shamim Mohamed, Ray Pereda, and Robert Parlett, "Programming with Unicon", http://unicon.sourceforge.net.

[8] Ralph E. Griswold and Madge T. Griswold, The Icon Programming Language, 3rd edition. Peer to Peer Communications, San Jose, 1997.

Scripting Runtime Dynamic Analyses

Jonathan E. Cook   Abdulmalik Al-Gahmi   Shalini Devi   Navin Vedagiri

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
jcook@cs.nmsu.edu

Abstract

Large scale system development and maintenance projects often need to build scaffolding—tools that help build the target system—that is customized to the project. For some classes of tools, including dynamic analysis, the cost barrier is too high to consider implementing customized support that might be beneficial to the project, and thus the project makes do with whatever off-the-shelf support is available.

This paper presents ideas and prototypes in offering generic support for high-level, flexible, and programmable introspection of software systems. Our hypothesis is that "quick-and-dirty" scripting languages such as Tcl/Tk and Python can be effectively used to create ad-hoc dynamic analyses that help system engineers better understand, develop, and maintain their system.

1. Introduction

Many system development and maintenance activities need or can benefit from introspective and possibly even manipulative capabilities in a running system. By this we mean the ability to peer into a running system and observe it, and even manipulate it to some extent. But typical mechanisms for introspection are hard to use, involve a great deal of low-level programming, and require expert programming to be used correctly. Because of this, the effort in building introspection tools is very high, and projects are often prevented from building application-specific tools or rapidly prototyping new general-purpose tools.

It would seem natural to provide some generic and easy-to-use mechanism to support these needs, and that is precisely the point of the ideas described here.

Our vision is to provide a flexible, easy-to-use mechanism for introspection that allows not just complex tools to be developed but allows the application programmers to easily build ad-hoc tools that meet a specific need at a specific time. Rather than try to predefine the capabilities *we* think might be needed, a better approach to achieve this end is to re-use one of the many scripting languages that are available.

Scripting languages allow extremely rapid development of functionality, at the the cost of speed since they are interpreted languages. But since they are full programming languages, there is no limit to the type of tools that might be built using them. While they do have some downsides, they seem ideal for building the scaffolding-type of software tools that must be built to help manage, test, observe, and maintain a large production software system.

In our initial prototypes we chose the Tcl/Tk scripting language because of its clean design, ease of integration with traditional programming languages (C/C++), and GUI capabilities. However, the principles underlying our approach can be applied using other languages.

2. Framework

Runtime issues in dynamic analysis have always had to balance the low-level issues of how to instrument the system under observation with the high-level issues of how to make the customization of analysis accessible to the user. A variety of solutions have been proposed and implemented, from special purpose systems that only allow a specific class of analyses to be performed, to special languages (e.g., event processing languages such as [1]) that can be used to specify the desired analysis.

All of these have their place; however, eventually one must consider that the scope of desired dynamic analyses is, in the most inclusive sense, general computation. Thus, why not enable general computational
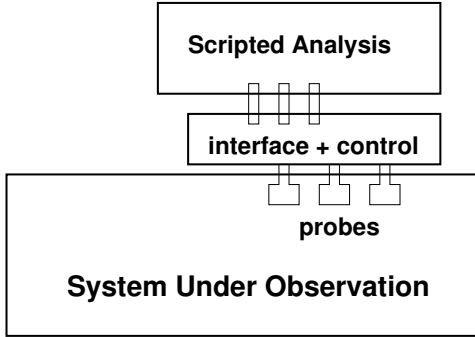
Figure 1. Scripted dynamic analysis architecture.

environments for dynamic analysis? Furthermore, can we make this programming of dynamic analyses more accessible by using high-level programming ideas?

Figure 1 shows how these ideas might fit together. The system under observation should have some mechanism, or at least the potential to insert a mechanism, for observing its behavior. The framework implementor can do the hard task of building the probes on top of this mechanism so that dynamic analysis tools can be built. To hide this complexity, the probe points and information they provide are made available to a scripting language engine, so that a specific analysis can be written in a scripting language, without needing to reach down into the details of the instrumentation.

Our initial prototypes, described in Sections 4-řefsec:java, have focused on method/function invocation interceptions, but our idea for the basic architecture is to enable script-level access to more types of instrumentation probes.

3. Tcl/Tk and other scripting languages

Tcl (Tool Command Language [8], pronounced "tickle") is a programming language in the class known as "scripting" languages. Newer scripting languages such as Tcl, Perl, Python, and PHP are much more advanced than the old shell scripting languages, yet they retain the ease of use and the capability for extremely rapid development of advanced functionality. Tcl and most other scripting languages can be both easily executed from C/C++ and extended with custom commands written in C/C++. It is rather misleading to call these languages "scripting" languages, in that they are very powerful interpreted languages, with built-in data structures and functional-style programming language constructs. Modules provide canned support for

web services, GUI interfaces, email, ftp, encryption, and many other high-level abstractions.

The upside of scripting languages is that one can create a great deal of functionality with relatively little effort, and they are robust enough to be relied upon. Indeed they can be found running much of the web services we use every day, are used extensively as the foundations of test harnesses, rapid prototyping environments, and many other real world situations.

The downside to most scripting languages is that they do not have a formal semantics but rather an operational one, which can change based on the version of the interpreter one is running! They are targeted towards achieving practical usefulness, not theoretical semantic correctness. However, compiled languages often reveal similar ambiguities [4]. Scripting languages are also quite a bit computationally slower than system programming languages, and their typically weak typing is sometimes detrimental.

4. Realization in CORBA

The CORBA (Common Object Request Broker Architecture) standard has defined cross-platform remote object invocation for ten years [7]. From early on CORBA had a proposed specification for object request interceptors, but it was incomplete and optional. Coincident with version 2.4.2 and later versions, a new interceptor specification was drafted, known as Portable Interceptors [2]. With the Portable Interceptor standard, it is now possible to create debugging, monitoring, and other introspection tools that will interoperate with most vendor ORBs.

In our work, we built an intermediate interceptor layer that took each interception point and invoked a mirror in the scripting language Tcl/Tk. Although not completely invisible to the application developer—some CORBA implementations may require rebuilding the application with different options—there is no low-level programming needed, and CORBA analysis tools that use interception-based data can be written completely in an easy-to-use scripting language.

CORBA Portable Interceptors are ORB-level interceptors that act upon method invocation requests and replies. On the server side, the most basic interception points are "receive_request" and "send_reply". Thus, CORBA interception points naturally give the analysis tool access to the pre- and post-execution points of an invocation. Also note that the interception points are generic for the ORB rather than specific to the object and method being invoked. Our interface makes available to the scripting language an object ID, the method name, and the values and types of the param-

eters and return values. Thus, at the script level, the analysis can perform computations specific to the object and/or method, by inspecting the meta-data.

5. Realization in Unix shared libraries

Shared or dynamic link libraries offer an interesting deployment opportunity for our ideas, because they are so widely used and their components are relatively simple (C functions). Nevertheless, the environment is one where very little meta-information is available, and with almost no runtime meta-programming ability. It has potential access, though, because we can modify the dynamic loading process to allow the possibility of binding a function call not to the original target function but to whatever we want, namely a probe point.

Once we have the call intercepted, it is "only" a matter of programming to implement the relay of the function call to the scripted dynamic analysis, and to ensure that the original function is still called, to effect the correct execution of the program. While not trivial, it is possible. Our work is currently in the context of the ELF object and library file formats [6], and in the Gnu shared library loader [5] as used in the Linux operating system.

When creating an object file which has calls to functions located in shared libraries, the compiler produces a call that uses table-based indirection (we will call them "jump tables"). In a somewhat simplified scenario, this table entry initially points not to the actual function (since its location is not known) but to the dynamic loader. Thus the first call invokes the dynamic loader, which will look up the function (by name), load the library if necessary, figure out the actual address of the function being called, overwrite the table entry with the actual function address, and then jump to the function. All subsequent calls from that call site simply pay a tiny (one instruction) penalty of a table lookup.

Although a static name interception capability already existed with the LD_PRELOAD environment variable, we have modified the Gnu dynamic loader to enable dynamic control of the name resolution process, for this and other work we are doing. The dynamic control allows runtime remapping of names to alternative names, on a per-link-object basis rather than at a global level. This functionality gives us the basic interception capability.

To avoid the necessity of low-level programming of the probes, we created a wrapper generator that takes function prototype definitions and generates probe wrappers that the dynamic linker can safely redirect the execution to. These wrappers also instantiate the argument and return data into Tcl-accessible data, and
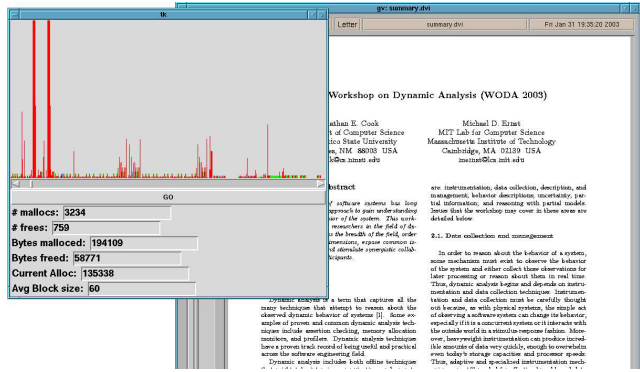

Figure 2. Memory usage analysis in the shared library framework.

invoke Tcl routines before and after the original function is invoked, so that pre- and post-execution analysis can be performed.

Thus, a dynamic analysis of an existing system can be written completely in Tcl, except for the function prototypes needed to generate the wrappers. Figure 2 shows an simple memory allocation analysis of an existing binary executable (Ghostview) that was written purely in Tcl/Tk.

An interesting difference between the interception points in this framework and in the previous one (CORBA) is that the C shared library interception points are specific to each function that is being intercepted. At the scripting level, a procedure must be defined for each pre- and post-interception point, for each function being intercepted. This is quite different than the generic interception point offered by the CORBA. The tradeoff between the two is that specific interception points offer more direct access to perform very specific ad-hoc analyses, while general needs such as event logging are much easier with the generic interception points (and the appropriate meta-data).

We have devised a mechanism for generic interception points in the shared library framework, but are still in the process of implementing it, and need to do more testing and make more meta-data available to define the actual interception.

6. Realization in Java

In both the CORBA and shared library environments we were dealing with compiled programs, and were able to utilize API's for the (compiled) interpreter of a scripting language (Tcl). In these settings, there is a clear distinction between the machine-code rep-

resentation of the system under observation and the interpreted language that the dynamic analysis tool is written in.

Java, however, presents an interesting case in that it is already an interpreted language, at least at the bytecode level. One might think that the Java environment, then, does not really benefit from having dynamic analysis tools able to be written in a scripting language. However, we feel that Java is a sufficiently complex language to warrant exploration of making dynamic analyses easier to program.

While Java can access native code resources, and thus could be integrated with external interpreters for scripting languages, there has been enough interest in the combination of Java and scripting languages that open source versions of Java-based interpreters exist for such popular languages as Tcl (Jacl, or Java Tcl) and Python (Jython). This means that we can have the scripted analysis running within the Java environment, which reduces complexity considerably, and future enabling of other probe points should be easier.

To create the interface between the system under observation with the scripting language, we have built a class wrapper generator which uses the Byte Code Engineering Library (BCEL [3]) to generate a wrapper for the public interface methods of a class. In this wrapper we generate calls to the scripting analysis program, with the appropriate data. Thus, the only code needed to be written by the developer interested in some ad-hoc dynamic analysis is the Tcl (Jacl) or Python (Jython) code. As with the C library framework, we offer pre- and post-execution points around the method call, and the interception points are specific rather than generic.

7. Conclusion

It is our hypothesis that developers would more often perform ad-hoc dynamic analyses on the system they are building or maintaining if the cost of creating these ad-hoc analyses was lower than it currently is. To this end, we are experimenting with enabling the analyses to be written in high level scripting languages, with the low-level details hidden and not needing to be the concern of the developers. We have built initial frameworks in CORBA using the Portable Interceptor standard, in C using the dynamic linking phase of library code access, and in Java using the BCEL toolset to access class bytecode. Each of these is centered around method or function call interception, but embody two different styles of access. The CORBA framework enables generic interception points, where all method calls fire the same interception points; while the shared

library and Java frameworks enable function/method-specific interception points. We plan to further enhance these frameworks and continue to explore the bounds of usefulness for scripting languages in dynamic analysis.

Acknowledgments

References

[1] M. Auguston, A. Gates, and M. Lujan. Defining a program Behavior Model for Dynamic Analyzers. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, pages 257–262. IEEE Computer Society Press, June 1997.

[2] Interceptors Published Draft with CORBA 2.4+ Core Chapters. Technical Report ptc/2001-03-04, Object Management Group, 2001.

[3] M. Dahm. Byte Code Engineering Library. 2002. http://jakarta.apache.org/bcel/.

[4] S. Eisenbach and C. Sadler. Changing Java Programs. In *Proceedings of the 2001 International Conference on Software Maintenance*, pages 479–487, Nov. 2001.

[5] Gnu C Library. 2002. http://www.gnu.org/.

[6] J. Levine. *Linkers & Loaders*. Morgan Kaufmann, San Diego, CA, 2000.

[7] OMG. The Common Object Request Broker: Architecture and Specification, v2.4.2. Technical Report formal/01-02-01, Object Management Group, 2001.

[8] J. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, Reading, MA, 1994.