

# Controlling Factors in Evaluating Path-sensitive Error Detection Techniques

Matthew B. Dwyer, Suzette Person, Sebastian Elbaum  
Department of Computer Science and Engineering  
University of Nebraska - Lincoln  
Lincoln, Nebraska  
{dwyer,sperson,elbaum}@cse.unl.edu

## ABSTRACT

Recent advances in static program analysis have made it possible to detect errors in applications that have been thoroughly tested and are in wide-spread use. The ability to find errors that have eluded traditional validation methods is due to the development and combination of sophisticated algorithmic techniques that are embedded in the implementations of analysis tools. Evaluating new analysis techniques is typically performed by running an analysis tool on a collection of subject programs, perhaps enabling and disabling a given technique in different runs. While seemingly sensible, this approach runs the risk of attributing improvements in the cost-effectiveness of the analysis to the technique under consideration, when those improvements may actually be due to details of analysis tool implementations that are uncontrolled during evaluation.

In this paper, we focus on the specific class of path-sensitive error detection techniques and identify several factors that can significantly influence the cost of analysis. We show, through careful empirical studies, that the influence of these factors is sufficiently large that, if left uncontrolled, they may lead researchers to improperly attribute improvements in analysis cost and effectiveness. We make several recommendations as to how the influence of these factors can be mitigated when evaluating techniques.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*assertion checkers, model checking*

## General Terms

Verification, Experimentation

## Keywords

Path-sensitive analysis, Model checking, Empirical study

## 1. INTRODUCTION

Static program analyses calculate information about the executable behavior of a program without running the program. Traditionally,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.  
Copyright 2006 ACM 1-59593-468-5/06/0011 ...\$5.00.

static analyses have been formulated to provide guarantees about program behavior to support, for example, semantics-preserving code transformations to improve performance. Such analyses must necessarily account for all possible program behaviors. In practice, this requirement forces analysis developers to formulate relatively imprecise analyses to achieve scalability to real programs.

It is also possible to formulate static analyses explicitly to detect errors and issue diagnostic information to users. The intuition behind such approaches is that a static analysis can be engineered to more efficiently cover a broader space of program behavior than can be achieved through testing and, consequently, such analyses have the potential to detect *hard to find* errors. Because they need not account for all possible behaviors, analysis developers have exploited this relaxed requirement to customize very precise path-sensitive analysis frameworks, for example model-checking tools like Bogor [23], Java Path Finder (JPF) [27], Mur $\phi$  [3], and Spin [17], to make them cost effective for error detection.

Several recent efforts along these lines involve adaptations of the CMC [20] model checker to make it more effective for finding errors in certain kinds of applications. In [19], the authors adapt CMC for error detection in protocol implementations and have used it to find four errors in the Linux TCP/IP implementation. More recently they have developed FiSC [28], a version of CMC that has been adapted for and used to reason about file system implementations; several significant errors in three widely-used file system implementations have been detected using FiSC. These adaptations have been carefully tuned to use specific heuristics for selectively storing only part of a program's data state during analysis and for prioritizing the order in which statements are analyzed.

Results such as these provide an important *proof of concept* that cost-effective and precise path-sensitive analyses for error detection can be built. They demonstrate that there exists a combination of specific techniques that can provide cost-effective analysis for a specific class of programs. They do not, however, provide information about the relative cost-effectiveness of the individual analysis techniques that they are comprised of nor about the range of programs over which they are effective. For example, two very different heuristics for prioritizing exploration of transitions are described in [19], preferring exploration of new behaviors relative to protocol states and preferring infrequent state changes, but information about the breadth or relative effectiveness of the heuristics is not provided. To be fair, this was not the goal of the authors, but it is important to gain this kind of information so that techniques such as these can be selected and combined for maximum benefit in path-sensitive error detection tools.

Obtaining such results requires careful empirical evaluation of techniques used to achieve cost-effective analysis across a range of programs. This kind of evaluation can be difficult to perform es-

pecially when there is a lack of knowledge about the factors that can influence the performance of an analysis tool. In this paper, we take a first step towards enabling controlled empirical studies of path-sensitive static analyses by presenting data on two factors that can *significantly* influence the performance of path-sensitive error detection analyses, to the extent that, if uncontrolled their influence may obscure differences in performance that are attributable to analysis techniques.

The first factor is related to the implementation of path-sensitive analysis algorithms. Most analysis algorithms allow the execution of a program to be under-specified in some way. When analyzing a multi-threaded Java program, for example, the analysis may reach a point where bytecodes in two different threads are enabled for execution. In a JVM, the thread scheduling algorithm will choose one of those bytecodes to execute first, but path-sensitive analysis techniques generally abstract from thread scheduling algorithm details and simply require that each of the schedulings is analyzed. Path-sensitive analysis tools, such as Bogor, Mur $\phi$ , SPIN, and JPF, implement a specific *default search order* for exploring simultaneously enabled execution steps; in fact, these four tools each implement different default orders. Given that these tools were built to exhaustively analyze all possible program paths the specific default order used was not a concern to their developers. When targeting or customizing such tools to detect errors, however, *we show that variation in search order can give rise to very large variations in path-sensitive analysis cost and fault detection effectiveness across a range of programs*. In Section 3, we support this conclusion with a *retrospective study* that looks back at previously published results and relates them to results from empirical studies we performed.

The second factor is related to the subject programs used to evaluate the cost-effectiveness of path-sensitive analysis techniques. The literature contains many papers that introduce analysis techniques and *illustrate* the performance of those techniques on a few small selected examples, for example, dining philosophers and bounded buffer examples [1, 6, 21]. Recent efforts to establish benchmarks to support the evaluation of testing and analysis techniques for multi-threaded Java programs are focused on making a broader collection of examples available to the community [10, 11]. One thing lacking from the literature and emerging benchmarks is a meaningful characterization of the programs and the faults they contain, and the criteria for their inclusion in the benchmark. This information would help researchers determine whether the benchmarks are appropriate to assess their particular techniques and, if they are appropriate, it would help them put their findings into perspective leading to claims that are substantiated in the collected data set.

We believe that there are multiple program factors that influence the cost of path-sensitive error detection techniques. For example, it has long been accepted that the degree of concurrency can dramatically influence the cost of analysis [1], but it may not always do so. The research community has not even identified a *complete* set of program factors that may influence path-sensitive analysis cost, much less characterized those influences. In this paper, we focus on two program factors: the *number of threads* and the *path error density*, the probability of a thread schedule exhibiting an error. Path error density was chosen because path-sensitive techniques are intended to detect *hard to find* errors in programs - thus it makes sense to assess the performance of techniques on examples that contain a variety of error densities. In Section 4, we describe a case controlled study that varies these two factors across a population of Java subject programs and analyzes the resulting analysis cost. We are able to conclude, from this study, that *programs with high path error density will exhibit almost no variation in analysis cost across*

*different path-sensitive analysis techniques*. Surprisingly, many of the examples currently used in evaluations in the literature have very high path error densities and our findings call into question the utility of those examples as experimental subjects.

To enable quantitative exploration of these issues, we set our work in the context of path-sensitive analysis of multi-threaded Java programs for detecting safety property violations. We use the JPF Java model checker as the basis for our evaluation of the influence of the factors described above, but we believe that our findings provide important information that can guide the evaluation of path-sensitive analysis techniques in general. The contributions of the paper lie in: (i) the identification of default search order as a factor that can impact the performance of path-sensitive error detection techniques; (ii) the identification of path error density as a program factor that can impact the performance of path-sensitive error detection techniques; (iii) the results of empirical studies that indicate the frequency and magnitude of performance variation with these factors (Sections 3 and 4); and (iv) recommendations for how to control for the effects of these factors in experimental studies (Section 5).

## 2. BACKGROUND

This Section gives an overview of the class of path-sensitive analyses realized by multi-threaded Java model checkers, explains the sources of search order variation in such analyses, and characterizes the availability of subject multi-threaded Java programs for experiments with path-sensitive error detection tools.

### 2.1 Path-Sensitive State-Space Search

Many path-sensitive analysis techniques treat programs as guarded-transition systems and analyze program behavior via depth-first traversal of transition sequences rooted at the initial state<sup>1</sup>. A *guarded transition system* consists of a set of variables, which for our purposes are coalesced into a single composite *state variable*  $s$ , and a set of guarded transitions which atomically test, with predicate  $\phi$ , the current state and update the state by executing a transition,  $\alpha$ , i.e., if  $\phi(s)$  then  $s = \alpha(s)$ . The initial values of program variables are used to define an initial state,  $s_0$ . Figure 1 presents the DFS analysis algorithm. On line 4, *enabled*( $s$ ) returns the set of transitions,  $\alpha$ , whose guard,  $\phi$ , is true in the given state. Lines 7-9 test if an error state has been reached, and if so, records the current DFS stack, which encodes the path under analysis, as a counterexample and exits. Even though this analysis does not generate all program paths, it is path-sensitive since it reasons about paths and prefixes of paths independently; a DFS can be thought of as analyzing all acyclic program paths.

Line 5 of this algorithm imposes no order on iterating through the set of enabled transitions in a state. This is an issue if non-singleton sets are produced at line 4 which is actually very common in analyzing realistic programs. This may seem odd since program execution is usually thought of as deterministic for a given execution environment, i.e., sequence of inputs and scheduling decisions. In practice, path-sensitive analyses must perform significant abstraction to gain tractability and to produce results that generalize across multiple specific execution environments.

For example, the past decade has seen a significant amount of work on predicate abstraction [13] which replaces reasoning about specific variable values with sets of values encoded symbolically. These abstractions encode approximations using non-determinism.

<sup>1</sup>We note that all of the search order issues we discuss for DFS are also relevant for other forms of search, such as breadth-first and variants of breadth-first search.

```

INIT
1  seen := {s0}
2  push(stack, s0)
3  DFS(s0)

DFS(s)
4  workSet := enabled(s)
5  for each  $\alpha \in \textit{workSet}$  do
6    s' :=  $\alpha$ (s)
7    if error(s') then
8      counterexample := stack
9      exit
10   if s'  $\notin$  seen then
11     seen := seen  $\cup$  {s'}
12     push(stack, s')
13     DFS(s')
14   pop(stack)
end DFS()

```

Figure 1: Depth-first search for first error state

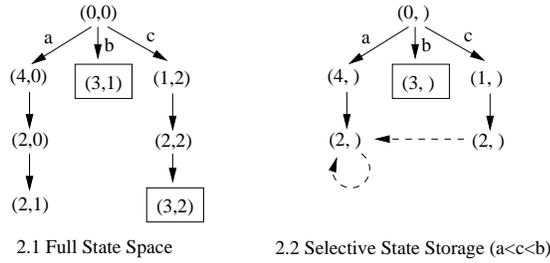


Figure 2: Search Order Example

For example, if a variable is abstracted by predicates  $x < 0$ ,  $x == 0$ ,  $x > 0$  then the result of executing a statement  $x = x - 2$  in a state in which  $x > 0$  could result in any of the three predicates being true and is therefore modeled with an enabled transition for each resultant predicate.

In reasoning about a multi-threaded Java program, a path-sensitive analysis must reflect the possible scheduler decisions that could be executed by a JVM. The JVM specification provides only a weak specification of scheduler behavior, i.e., higher priority threads should be scheduled first. It says nothing about the order in which threads at a given priority level should be executed; in fact, it does not even require that threads be executed fairly. All JVMs implement a scheduling policy and that policy calculates a thread ordering. To provide a degree of JVM independence, path-sensitive analyses for multi-threaded Java programs use non-determinism to over-approximate the set of all legal JVM scheduler decisions. Transitions in multiple threads will typically be enabled in a given state and returned on line 4 in the algorithm above. Non-determinism has other uses in tools such as JPF as well; for example, to generate a range of input values [14] or to encode complex specifications [2].

These sources of non-determinism combine to produce a large space of possible paths through the transition system. When performing an exhaustive traversal of the transition system states, the order in which transitions are executed at line 4 is of no consequence; the algorithm is guaranteed to visit all reachable system states regardless of order. Clearly, when an error exists order may influence the cost of the search. Intuitively the presence of non-determinism is the key factor in determining the number of paths to be traversed; a deterministic system has a single path. To illustrate, consider Figure 2.1 where a search order of  $b < a < c$  finds an error, denoted with a rectangle, after exploring 2 states. If search order  $a < c < b$  is used, then 7 states must be explored.

## 2.2 Heuristic Program State-Space Search

Recent years have seen a growing interest in the incorporation of heuristics into path-sensitive analyses. Heuristics are typically designed to calculate a search order that will reach an error state quickly, e.g., [14], or to reach a particular kind of goal state, e.g., one with a short counter-example [8, 26]. We note also that heuristics can choose to completely drop transitions from consideration [19], which clearly changes the search order.

Heuristics can function in several ways. In traditional heuristic search, one applies a *cost* function to map each enabled transition,  $\alpha$ , to a value and then implements line 5 of Figure 1 to iterate over the transitions in cost-order. Cost functions usually calculate a score based on the current state in the exploration or based on the path explored up to the state, for example, Groce and Visser’s [14] “demonic” scheduling heuristic scores thread transitions based on the frequency of thread execution along the path.

It is important to observe that using this type of heuristic does not completely eliminate the issue of transition order. If two transitions evaluate to the same cost value, a common situation when using discrete cost functions [14], then the order in which those transitions execute is left to the stability of the transition sorting algorithm used to implement line 5. A stable sort will honor the underlying order implemented in the model checker, whereas an unstable sort may modify it. Thus, two different heuristic path-sensitive analysis implementations that use the same cost function may actually explore the paths in the system in a different order.

Another heuristic approach is to be selective in storing the program state. This is realized by modifying line 10 of Figure 1 so that the membership test is not performed on the complete state,  $s$ , but rather on a projection of the state  $\pi(s)$ , and projected seen values,  $\{\pi(s') \mid \exists s' \in \textit{seen}\}$ . For example, Musuvathi and Engler [19] drop a variable from the state if it has been assigned a large number of distinct values on the path explored. This has the effect of forcing backtracking in the DFS earlier than would happen without this modification. In doing this, the search order may change since the continuation of the current path in the original DFS is either eliminated from the search or deferred until later in the search.

Figure 2.2 illustrates a selective state storage strategy where the second state component is dropped. Under the order  $a < c < b$  one can see that the traversal of the state space is curtailed prematurely, potentially reducing the analysis cost, but in this case it forces the error state along the path beginning with transition  $c$  to be missed due to matching on the partial state  $(2, \cdot)$ , denoted with a dashed arrow. The net result is that the search still requires exploration of 7 states to find an error.

Heuristics are viewed by many as a promising mechanism for mitigating the combinatorial explosion in the cost of path-sensitive state-space analyses. They have the effect of focusing the search on a portion of the state space. Unlike property-preserving state-space reductions, heuristics are oriented towards error detection and the only valid means of evaluating them is through broad experimentation across a variety of programs and properties. Consequently, *we believe that evaluation of heuristic state-space search techniques is especially vulnerable to a lack of control on experimental factors.*

## 2.3 Java State-Space Analysis Tools

In our studies, we consider a single path-sensitive analysis tool, JPF, since it is one of the most mature and sophisticated path-sensitive multi-threaded Java analysis tools that we are aware of, it is used widely by researchers, and it provides a clean implementation that makes it easy to modify the transition order.

JPF is built as a virtual machine that stores the set of states visited along a path and uses that state set to force backtracking along

other paths as in the general DFS algorithm above. JPF processes JVM bytecode programs directly and as a result it relies on a Java compiler to translate source programs. JPF has a flexible architecture that makes heavy use of Java interfaces and abstract classes to consolidate common functionality in the model checker and enable different algorithms and data structures to be used.

The `Search` interface defines the generic API used by the main analysis module. Multiple search modules are included with JPF including: `DFS` - depth-first search, implementations of all of the heuristics described in [14], and `RandomSearch` - a stateless search that explores a single path in the program. The `Scheduler` abstract class is the base class used to define the strategy for calculating the order in which enabled threads are analyzed; this corresponds to line 5 in Figure 1. JPF includes two `Scheduler` sub-types: `DefaultScheduler` - which implements a fixed strategy for selecting the *next* thread based on the order in which `Thread` (or `Runnable`) objects are allocated along the path being analyzed, and `RandomOrderScheduler` - which randomizes the order of thread selection based a pseudo-random sequence determined by the current time or a user-specified seed value.

An additional degree of *non-determinism* can be specified in JPF programs, for example, a call to `Verify.random(3)` returns one of  $\{0, 1, 2, 3\}$ ; in a complete stateful search all such values are guaranteed to be returned. The order in which these values are produced is up to the implementation of this method; the current implementation produces them in their value order. Calls such as this create multiple enabled transitions *internal* to a single thread.

Version 3.1.2 of JPF does not allow for randomization of the order in which such internal transitions are explored; a forthcoming version of JPF will have this feature. Given this, the results we report for analyses using JPF with `RandomOrderScheduler` should be interpreted as exploring a subset of all possible random orders. The Java programs we consider in our studies have very limited internal non-determinism, so while we believe that additional variation in search order may be possible when running JPF on those example we regard it as a minor effect. Even if that were not the case, the results we report on the variation in performance of JPF due to search order can be regarded as a lower-bound.

The only modifications we made to JPF were to allow for time-bounded analysis, a command-line parameter specified the maximum number of seconds an analysis may execute, and the reporting of statistics on partial searches that are terminated either when the time bound is reached or memory is exhausted. Neither change affects the path-sensitive analysis implementation.

Version 3.1.2 of JPF has not, to the best of our knowledge, been used in any published study of error detection techniques. Previous papers reporting JPF results, such as [14, 21], used older versions that were missing important advances in mitigating the cost of path-sensitive analysis. For example, canonical heap symmetry reductions, which represent all execution states of a Java program that differ only in the physical addresses of objects using a single representative state, e.g., [24], are implemented. In addition sophisticated partial order reductions that are customized for multi-threaded Java programs [7], have also been adapted to JPF and implemented. The cost-effectiveness of these reductions is sufficient to regard the use of these features as the default configuration of JPF. All of our runs use these features. As a consequence, even on the same Java program the performance measures we report may differ significantly from those reported in previous studies.

## 2.4 Default Transition Search Order

Default transition search order is an *implementation detail* that is

typically realized in a way that is convenient given the internal path and state representations maintained by an analysis tool. It is no surprise then that the default transition order varies between tools. In fact, each of Spin, Mur $\phi$ , JPF and Bogor use a different default order to select the next enabled thread to be analyzed. Spin selects the next thread based on the order in which `active` proctypes, i.e., Spin's notion of thread, appear in the source file and then in the order in which dynamically started threads are created. Mur $\phi$  orders threads in the reverse order of their appearance in the source file. JPF and Bogor are very similar; JPF uses the strategy described above for the `DefaultScheduler`, whereas Bogor uses the order in which the thread `start()` method is invoked. Despite the close similarity in implementations, certain program structures can give rise to dramatically different default search orders. For the following code:

```
Thread[] threads[3];
for (int i=0; i<3; i++) threads[i] = new Thread();
for (int i=0; i<3; i++) threads[3-i].start;
```

JPF and Bogor will explore the threads in opposite orders.

## 2.5 Common Multi-threaded Java Subjects

The past decade has seen significant advances in the development of path-sensitive program analyses. There have, however, been relatively few broad evaluations of the cost-effectiveness of those techniques across a range of systems; a notable exception is Corbett's study of deadlock detection in 30 different multi-tasking Ada programs [1].

For analyzing multi-threaded Java programs, researchers have had two basic choices: (1) adapt existing examples from the concurrency literature, or (2) target real multi-threaded Java programs. Most have chosen option (1), since it can be difficult to obtain faulty versions of real multi-threaded Java programs, even from open source projects, and because using a small set of well-understood examples would seem to provide a means for comparing performance across tools and techniques.

We obtained the suites of programs compiled by the Bandera [6] and JPF projects. These programs cover nearly all of the multi-threaded Java programs that are used in evaluating path-sensitive Java analyses in the literature; some papers have also used Java standard library implementations as analysis subjects. We selected programs that exhibit some type of concurrency error. These examples can be divided into two kinds: concurrency error *kernels* and *realistic* programs. Concurrency error kernels are very simple programs that distill the essence of a particular concurrency error. Examples include adapted versions from the concurrency literature, such as dining philosophers, as well as programs that exhibit Java-specific errors; we had a student independently implement kernels for the Java *concurrent bug patterns* (CBP) described in [12]. These kernels typically include the control and data structures required to exhibit the error and nothing else. Realistic programs are small to medium size programs that perform a computation over rich data structures. They tend to be much larger than the concurrency kernels, often accept input data that parameterizes the computation, and include significant control and data structures that are unrelated to the error.

We were also granted access to a collection of multi-threaded Java programs being developed at IBM to support testing and analysis research [11]. This set of programs overlaps with the Bandera and JPF programs to some extent, but it also includes a number of programs that were developed to encode common Java concurrency bug patterns; we refer to those programs as the IBM *benchmarks*. Many of the IBM benchmarks were written following standard forms for parameterization, e.g., the degree of multi-threading

Subject	Source	Reference	Kind	Parameters	Error	Classes	SLOC
Account	IBM	[10]	benchmark	none	Deadlock, Race	3	66
AccountSubtype	IBM	[10]	benchmark	none	Race	6	91
Airline	IBM	[10]	benchmark	#ticketsIssued, cushion	Race	2	31
AlarmClock	Bandera	[6]	real	selector	NullPtrExcpn	6	125
AllocateVector	IBM	[10]	benchmark	blockSize, vectorSize, #runs	No Lock	3	85
BoundedBuffer	Bandera	[21, 6]	real	size, #prods, #cons, modCount	Deadlock	5	65
Clean	CBP	[12]	kernel	#first, #second, #iter	Deadlock	4	51
Daisy	other	[22]	real	none	Assert	21	744
Deadlock	Bandera		kernel	selector	Deadlock	4	24
DEOS	JPF	[14]	real	abs?	Assert	24	838
DiningPhil	Bandera	[14]	kernel	selector, #forks	Deadlock	3	25
Elevator	other	[9]	real	none	ArrayIdxOOBExcpn	12	934
LinkedList	IBM	[10]	benchmark	#builders, maxSize, synch	Atomicity	5	117
LoseNotify	CBP	[12]	kernel	#waiters, #notifiers, #iters	Deadlock	4	41
NestedMonitor	Bandera		kernel	none	Deadlock	6	53
Piper	IBM	[10]	benchmark	#seatRequests, #passengers, capacity	Deadlock	2	71
ProducerConsumer	Bandera		kernel	#prods, #cons, #items	Race	8	87
Reorder	CBP	[12]	kernel	#setters, #checkers	Atomicity	4	44
ReplicatedWorker	Bandera	[21, 6]	real	#workers, #items, min, max, epsilon	Deadlock	14	304
RaxExtended	JPF	[21, 14, 6]	real	gc, wc, envFirst?	Race	11	127
RW	Bandera	[21, 6]	real	#readers, #writers, bound	Deadlock, Race	6	103
SleepingBarber	Bandera	[6]	kernel	none	Deadlock	4	66
TwoStage	CBP	[12]	kernel	#twoStagers, #readers	Two-stage Access	5	52
WrongLock	CBP	[12]	kernel	#dataLockers, #classLockers	Wrong Lock	4	38

Table 1: Subject Program Descriptions

in an example was indicated by a string `little`, `average`, or `lot`, for error reporting, e.g., error messages were printed to a log file, and for perturbing the schedules so as to make errors more difficult to detect by testing, e.g., by inserting random `sleep()` calls. We transformed these examples to further parameterize them, e.g., programs accept an integer that indicates the degree of multi-threading, to indicate errors through assertion violations, and to remove `sleep()` calls; this last step is easily automated. We performed one example transformation that was non-trivial. This involved refactoring the `Account` example into a version, called `AccountSubtype`, that uses an interface and sub-typing to organize different types of accounts, e.g., checking, savings, etc. The rationale for this was that it is a more realistic object-oriented structure for this program and the fault was related to account type, so this might help us understand the interaction of between OO structure and fault detection. In all cases, the resultant versions of the IBM benchmark programs we considered are behaviorally equivalent to the original versions.

Table 1 lists 24 subject programs and describes their source, kind (i.e., kernel, realistic, benchmark), parameters, errors - including CBP designations where appropriate, application class counts, and non-comment source lines of code as calculated by `JavaNCSS` [18]. Note that the `selector` parameter is used to select from an existing set of subject variations available in the suite of examples available with `Bandera`. Errors other than deadlock or exceptions are triggered by explicit `assert` statements in the subject programs. For subjects, with more than one kind of error the suffix `"NoDeadLockCk"` indicates that deadlock checking is disabled, so that the other error can be detected; a suffix `"NoExceptCk"` indicates the symmetric case. We note that the analysis of these programs considers all of the library code used by the applications which can significantly increase analyzed program size.

### 3. DOES ORDER REALLY MATTER?

It seems obvious that search order *can* matter, but the real ques-

tions are: (a) *Can search order cause performance to vary enough to affect the conclusions of carefully performed evaluations?* and (b) *Does order matter across a range of programs?* (as opposed to toy examples like the one shown in the previous section). In this section, we provide anecdotal evidence regarding question (a) and then followup with a broader study to assess question (b).

#### 3.1 An Anecdote

Even if every path-sensitive analysis of every program were susceptible to the influence of varying search order, we would not be concerned if that influence was small. Path-sensitive analyses can be quite expensive and the community expects that techniques of practical importance will yield significant improvements; we need not be too concerned with small scale effects.

To assess question (a), we considered the results of an existing carefully performed study that compared four path-sensitive analysis tools on a number of versions of models of the GNU implementation of the UUCP i-Protocol [5]. We downloaded the `i3` and `i4` `Murφ` models from the author's web-site and configured them to be `2fn` models as described in their study. The purpose of their study was to understand the performance improvements that could be achieved by applying five different abstractions to the model. The authors were able to order the abstractions based on analysis performance in finding errors in versions of the system; they also considered performance in showing the absence of errors in versions that were free of errors. They determined that `i4` was uniformly faster than `i3`.

We modified the implementation of `Murφ` version 2.70L to randomly choose the order in which enabled transitions are explored during the search of system paths for violations of safety properties; the random order was seeded by system time when the analysis was initiated. We executed `Murφ` on the models using the default search order of the tool and 20 different randomized orders. Since we ran on a much faster platform, the execution times are not comparable to the results published in [5], so we compared runs with default search order to runs with randomized orders using the state count

Subject	New States					Found		Retrospective Study	
	Default	Average Random	95% CI Random	Minimum Random	Maximum Random	Default	Num. Random	Technique	Speedup Ratio
Account-NoExcepCk	34484	1097737	±168942	92	8393836	OM	86		
Account-NoDeadLckCk	1034	695726	±128737	257	8041901	✓	495		
Airline-20.2	6942672	298274	±101032	148	4842487	TO	469		
AlarmClock	290	264	±8	83	438	✓	500	slice[6]	1.28
AllocateVector-2.100.1	20352435	133908	±11448	43	484132	TO	500		
BoundedBuffer-2.4.4.1	802542	27272	±2167	147	132771	✓	500		
Daisy-4.3	101816	70356	±2750	35256	99774	✓	500		
DEOS	260039	506816	±22712	103065	1213707	✓	500	A*[14]	3.12
DEOSAbstracted	54	783	±67	7	2877	✓	500	choose-free[21]	1.54
DiningPhil-8	1652	38	±1	17	131	✓	500	most-blocked[14]	29.41
LinkedListNoSynch-4.100	60964	7972	±55	5866	9448	✓	500		
Piper-2.16.8	6377527	3739112	±95781	402	4905845	TO	43		
ReplicatedWorker-5.2.0	6231840	4670826	±186362	485	6485120	TO	85	choose-free[21]	1.0
RaxExtended-4.3	3470398	195492	±33106	46	3108548	OM	497	interleave[14]	2.44
RW-2.2.100NoDeadLckCk	3147356	13619036	±1036042	49	24225772	✓	214	choose-free[21]	384.62
SleepingBarber	36	34	±1	28	39	✓	500	slice[6]	1.15

**Table 2: Comparative summary of random versus default search strategies.**

measure used in original the study. We found that the default order for *i4* explored 218 states and that there was an order of search for *i3* that only explored 87 states. Similarly, the default order for *i3* explored 397 states and there was an order for *i4* that explored 703 states. The point is that the variation in performance due to search order for  $\text{Mur}\phi$  on these problems is sufficiently large so that, in some cases, the conclusions about the cost-effectiveness of abstractions *i3* and *i4* that were originally drawn by considering default orderings would be inverted.

Based on this limited experience, we conjecture that variations in search order may invalidate the results of otherwise carefully performed evaluations of path-sensitive error detection techniques.

## 3.2 A Retrospective Study

To evaluate this conjecture more broadly, and thereby address questions (a) and (b), we divide this study into two parts. First, we quantify the variation in performance we observed when running JPF to find an error using randomly chosen search orders on a set of selected subjects. In the second part of our study, we relate the variation we observed on subjects utilized in previous studies back to the results reported in the those studies.

### 3.2.1 Dependent Variable

In this study, we measure the dependent variable, performance, in terms of the number of *new program states* explored during the analysis. This measure is commonly used in model checker evaluations. It is also system independent, making it possible to compare analysis performance across platforms which for our study is crucial since we do not have access to the execution platforms used in previously reported studies.

### 3.2.2 Independent Variable

Our study manipulated one independent variable: the search order. Given the differences we encountered in default search order across existing path-sensitive analysis tools, we believe it would be difficult to characterize the space of all *implementable* orders that an analysis tool developer might choose. Consequently, we chose to randomize the search order. For each subject, we executed JPF configured with depth-first search, using the `DFS` component, and we selected either the `DefaultScheduler` or the `RandomOrderScheduler` as described in Section 2.3.

### 3.2.3 Study Design and Setup

For both parts of this study we used the following subjects from Table 1: `AlarmClock`, `DEOS`, `DEOSAbstracted`, `DiningPhil`, `ReplicatedWorker`, `RaxExtended`, `RW`, and `SleepingBarber`. Each of these programs appeared in one or more of [6, 14, 21].

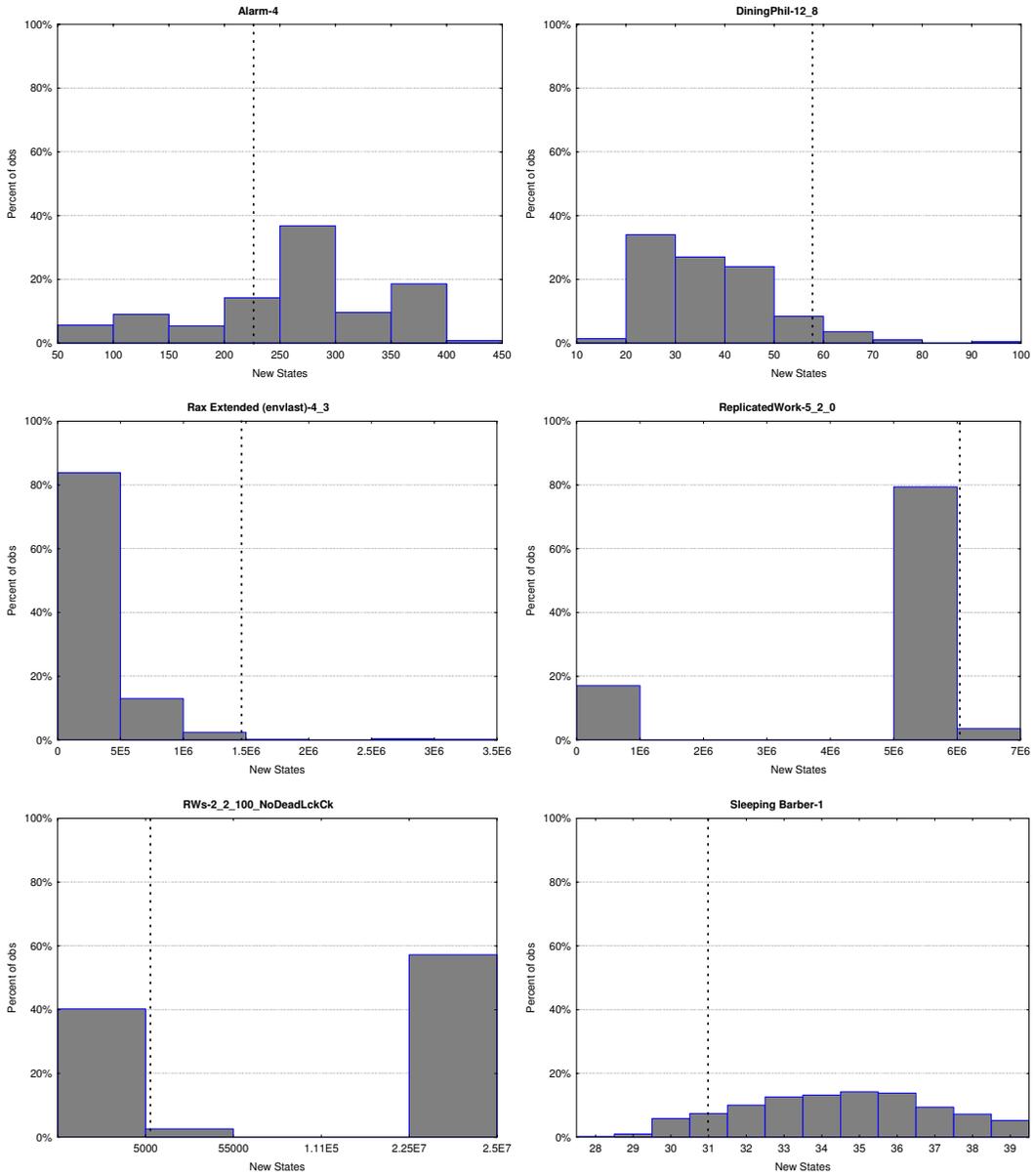
In the first part of our study, we also included all non-kernel subjects from Table 1 in order to assess the variation in analysis cost on the more *realistic* subjects. Since these examples were not the subject of previous studies we do not include them in the retrospective part of our analysis. Given the large number of subject programs to choose from, we also considered a set of secondary factors when selecting our subject population. These factors include the size of the program (in terms of lines of code and thread counts), the type of concurrency fault, and inclusion of the non-trivial code bases used in evaluating path-sensitive error detection techniques that we are aware of and have access to. Our goal was to choose a variety of programs to increase the diversity of our subject population.

In the second part of this study, we relate the variation in observed performance of JPF back to the results reported in the three selected studies. Each of the studies quantified the benefit of a proposed analysis technique by comparing the cost of analysis using the technique to the cost of analysis using the default search order. We define the *speedup ratio* of these analysis costs as:

$$\text{speedup ratio} = \text{default order cost} / \text{technique cost}$$

A speedup ratio greater than one indicates that the technique showed an improvement over the default search order.

For example, [6] compares data on the performance of an analysis that uses program slicing as a pre-processing step to analysis without slicing; slicing can drop statements from the program and thus can change search order. In that study, the authors report a speedup ratio of 1.28 for one Java program thus indicating that slicing provides a non-trivial reduction in analysis cost. For our study, we calculated the speedup ratios for all of the techniques and subject programs used in the three previous studies; all such ratios are greater than or equal to one, since the techniques studied improve on the default search order. Our goal is to determine whether varying the search order could invalidate the conclusions drawn about the improvement of *any* one of the previously reported techniques, relative to the default search order. To simplify the presentation of our results, rather than discussing all of the previous techniques and subjects in our data analysis we only discuss the



**Figure 3: Scaled Default vs. Random Search Order**

technique and subject with the lowest speedup ratio. The selected technique, study and speedup ratios are shown in the rightmost two columns of Table 2.

The version of JPF used in the previously reported studies is no longer available. To enable our retrospective study, we translated the observed analysis cost in those studies to a cost measure for the JPF 3.1.2 implementation. For a given technique, we define the *translated technique performance (TTP)*:

$$TTP = \text{default order cost} / \text{speedup ratio}$$

using the speedup ratio calculated for that technique and the default order cost for JPF 3.1.2.

Finally, we compared the translated technique performance to the variations in performance we observed over the range of orders considered in our study. Orders that result in lower analysis cost

than the TTP indicate that using a different default search order in the original study could have possibly led the authors to draw a different conclusion about the benefit of the technique they studied. In such cases, we say that the variation in performance due to search order is of *practical significance*.

To perform this study and the follow-on study discussed in Section 4, we compiled all subjects using Java v1.4.2.07 and then model checked each subject using JPF v3.1.2 with partial order reductions enabled. The study was performed on a cluster of dual-Opteron 250's running at 2.4 GHz with 4 GByte of memory and running Fedora Core 3 Linux. Each subject was model checked one time using JPF's `DFSearcher` and then model checked 500 times using JPF's `RandomOrderSearcher` using system time as the seed. JPF's execution time was limited to one hour of wall time for all runs and all subjects; this limit exceeds most of the time-bounds used in previous studies (except for [6]).

### 3.2.4 Results and Analysis

We start by providing a comparative summary of the performance of search orders in terms of new states visited for the 16 subjects we considered in this study. This summary is presented in Table 2 which includes the values for the default search order, and the min, max, average and 95% confidence interval for the random order search based on the 500 observations collected for each subject. We also include information for both the default and random runs indicating if the error is found. *OM* indicates the default search order ran out of memory, *TO* indicates the run timed-out, and *Num. Random* represents the number of random runs (out of 500) that found the error. Note that where a default search did not find the error, the number of states listed can be considered a lower bound on the number of states that would be traversed if the search had been allowed to run to completion. The final two columns in Table 2 show the technique and the speedup ratio calculated based on the results in the previously published studies for those subjects. Note that the names used in this table, and in subsequent text, are the program name from Table 1 with a ‘.’ separated list of parameter values for the program following a ‘-’.

When observing the reported number of new states traversed by the different search orders, we first note the great variability across subjects; the number of new states reported ranges from the tens to the millions. This is probably not surprising given that our subjects vary significantly in the number of lines of code, number of threads and the general complexity of their control and data structures. We can also see that the default search order visits more states than the average random search in 10 of the 16 subjects. Interestingly, none of the default runs reported new states within the 95% confidence interval computed based on the 500 random runs.

Variability in results between the default search order and the random order for a given subject is also of interest. For example, the default search order for `AllocateVector-2.100.1` explores over 20 million states without finding the error while all of the random runs find the error in an average of 133,908 states (in fact, all of the 500 random order searches find the error in under .5 million states and in as few as 43 states). On the other hand, the default search order for `RW-2.2.100NoDeadLckCk` finds the error in 3.1 million states versus an average of 13.6 million states for the random runs, of which only 214 of 500 find the error. Perhaps, even more important than the number of states traversed is whether a search actually succeeds in locating the error when resources are bounded. Interestingly we note that in the six subject where the default search order does not find the error, at least 8% of the corresponding random searches for each program finds the error, and in half of these programs, over 80% of the random searches find the error.

Clearly, researchers utilizing programs reporting a smaller number of new states such as `AlarmClock` are less likely to be able to discriminate or expose the potential of their error detection technique. However, targeting programs with a larger state space is also challenging because of the variability we observe with the analysis of such programs. For example, for `RW-2.2.100NoDeadLckCk`, the 95% confidence interval around the mean covers a range of over one million new states. These observations attest to the degree of variability observed when program paths are traversed in different orders, and they also emphasize the importance of properly qualifying findings when evaluating a path-sensitive error detection technique relative to the single default order implemented in an analysis tool, since, as mentioned in Section 2, default order varies across tool implementations.

Figure 3 shows histograms for six of eight subject programs re-

lating the TTP to the performance of 500 randomized analysis runs. Two subjects are not shown: `DEOS`, discussed below, and `DEOSAbstracted` since it is very similar to the plot for `SleepingBarber`. The x-axis ranges from the minimum to the maximum number of new states across all analyses and is partitioned into regions. The y-axis shows the percentage of the 500 random runs whose performance lies in each of the regions. The dashed line is the TTP for the selected technique.

For one subject, `DEOS`, the techniques previously reported improved analysis enough relative to the default search order that they overwhelmed any variation in cost due to randomization observed in our study. For three of the subjects, `DiningPhil-8`, `RaxExtended`, and `ReplicatedWorker`, more than 86% of the 500 random order searches are classified as having practically significant variation from the TTP. For the remaining four subjects, the percentage of practically significant orders varies from 7% to 40%.

We believe that these findings tell a strong cautionary tale. Even for carefully planned and conducted studies of path-sensitive error detection techniques, failing to account for the influence of default search order exposes researchers to the possibility that the reported benefits of techniques are attributable to default search order rather than the technique itself.

## 4. WHAT PROGRAM FACTORS INFLUENCE ANALYSIS COST?

In this Section, we address the question: *What characteristics of programs cause significant increase in the cost of path-sensitive error detection techniques?*

### 4.1 Two Candidate Factors

The model checking community, in general, believes that there is a strong correlation between the number of threads in a concurrent program and the number of reachable program states. Holzmann, the author of the SPIN model checker, notes that: “In the worst case, the global reachability graph has the size of the Cartesian product of all component systems. ... Although, in practice, the size of the global reachability graph never approaches the worst case size, the reachable portion of the Cartesian product can also easily become prohibitively expensive to construct exhaustively.” [17]; a “component system” in SPIN is analogous to a thread in Java. While careful studies of the cost of model checking and related path-sensitive analyses are rare, the few that exist, such as Corbett’s [1], suggest that in practice, analysis cost grows exponentially with the number of threads. This lead us to consider the number of threads as factor that can strongly influence analysis costs.

Our second factor attempts to capture the intuitive notion of *hard to find* bug that is often used to characterize the sweet spot for path-sensitive error detection techniques, e.g., [20, 28]. Testing researchers have studied a number of measures for characterizing the ease with which a fault in a program can be revealed. For example, Hamlet and Voas [15] defined the notion of program testability as “... the probability that if P contains fault(s), P will fail under test.” In programs with high testability, faults are likely to be revealed as failures, while programs with low testability are unlikely to expose their faults. Unfortunately, the existing body of work on sensitivity analysis and testability has not explicitly considered concurrent or multi-threaded programs. In such programs there is an additional *input* that can lead to faults being exposed or hidden - the thread schedule. Rather than extend existing testability notions to account for scheduling decisions, we fix the program inputs to isolate thread scheduler decisions as the only varying *input* to the program under analysis. By randomly sampling from the set of possible thread

Density	Thread Count		
	<5	5..10	>10
<0.1	AllocateVector(2,100,1):<3, .038> Clean(1,1,12):<3, .009> Daisy():<3, .0007> DEOS(false):<4, 0.0> Elevator():<4, 0.0> TwoStage(1,1):<3, .016> Reorder(1,1):<3, .002>	BoundedBuffer(2,4,4,1):<9, .027> LinkedListSynch(4,100):<5, 0.0> Piper(2,4,4):<9, .073> TwoStage(2,2):<5, .012> TwoStage(2,5):<8, .019> Reorder(1,5):<7, 0.0>	BoundedBuffer(3,6,6,1):<13, 0.0>
0.1 . . . 0.9	AlarmClock(4):<4, .232> AlarmClock(9):<4, .227> AllocateVector(8,20,1):<3, .75> AllocateVector(2,20,4):<3, .575> AllocateVector(2,20,1):<3, .204> Deadlock(1):<3, .754> Deadlock(2):<3, .638> DEOS(true):<4, .415> WrongLock(1,1):<3, .406>	Account-NoExcepCk():<6, .114> Account-NoDeadLckCk():<6, .663> AccountSubtype(8,1):<10, .526> Airline(6,1):<7, .657> Airline(6,2):<7, .836> ProducerConsumer(2,2,4):<5, .146> ProducerConsumer(2,4,4):<7, .294> ReplicatedWorker(5,2,0.0,10,7,0.05):<6, .261> ReplicatedWorker(8,2,0.0,10,0,0.001):<9, .703> RWNoDeadLckCk(2,2,100):<5, .495> RWNoExcepCk(2,2,100):<5, .433> RaxExtended(2,3):<6, .761> RaxExtended(4,3):<6, .794>	Airline(20,2):<21, .757> Airline(20,8):<21, .809> Clean(10,10,1):<21, .477> Piper(2,8,4):<17, .336> Piper(2,16,8):<33, .118> ProducerConsumer(2,8,4):<11, .244> WrongLock(1,10):<12, .712> WrongLock(10,1):<12, .642>
>0.9	LoseNotify(1,12,18):<4, 1.0> NestedMonitor():<3, 1.0>	Clean(2,2,24):<5, .958> DiningPhil(12,6):<7, 1.0> DiningPhil(12,8):<9, 1.0> LinkedListNoSynch(4,100):<5, 1.0> SleepingBarber():<5, 1.0>	AccountSubtype(5,5):<11, .996> BoundedBuffer(2,10,1,1):<12, .996> Clean(10,10,120):<21, .994> DiningPhil(12,10):<11, 1.0> LoseNotify(10,10,120):<20, 1.0>

**Table 3: Parameterized Subject Instances with Measures**

schedules and checking for errors, we generate an estimate of the probability that a thread schedule for a program run will exhibit a failure. We call this probability estimate *path error density*, and use it in our studies as a surrogate-measure to capture the notion of the difficulty associated with exposing a bug.

We then conjecture that the number of threads used during program execution and the path error density of a program are important factors in determining the cost of path-sensitive error detection techniques.

## 4.2 A Case Controlled Study

To evaluate this conjecture, we performed a case controlled study. In this type of study, a researcher identifies groups of subjects with different characteristics that are of interest, and then analyzes the relationship between their characteristics and one or more dependent variables. In our study, we focus on the potential effect of the number of threads and path error density on the dependent variables: number of new states, error depth, and whether or not an error is found.

### 4.2.1 Characterization Variables

The characterization variables in our study are (1) the total number of threads created during the execution of the subject, and (2) the path error density. The first is simply a count of the main thread plus any child threads created during execution. The second measures the probability of finding a schedule that exhibits the error. We calculate path error density using JPF configured with the `RandomSearcher` and `RandomOrderScheduler`. This has the effect of simulating a single run of the subject program making a randomized sequence of scheduler decisions. We ran between 1000 and 10000 such runs for each subject and report the observed probability that an error is found on a given run in this sample as the path error density (the number of runs depended on the variation we observed in the collected sample). We note that for some subjects, we were unable to detect an error in 10000 runs and we

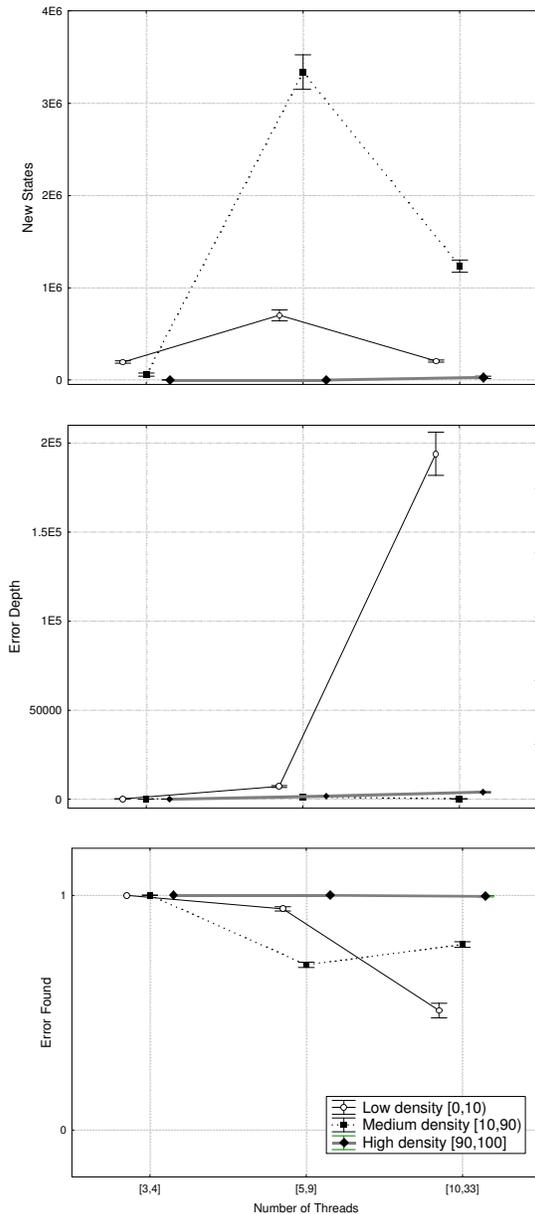
(under) estimate their error density as 0, since we know that they contain an error. Note that some of the kernel subjects contain infinite loops; for those we ran the simulation for a bounded number of steps where we selected a bound that was significantly greater than, i.e., four times, the shallowest error depth encountered for that subject to ensure we would detect a diversity of paths with errors.

### 4.2.2 Study Design and Setup

To conduct this study, we required a sample of programs that exposed a range of values for the characterization variables so that we could begin to understand their effect on the dependent variables.

To systematically obtain such a sample, we designed a two dimensional matrix, where each dimension was given by a characterization variable (path-density and thread count). We then partitioned each dimension in three categories. Based on “rules of thumb” in the analysis community we chose thread count categories of  $\{< 5, [5, 9], > 10\}$ . Since we did not have such rules of thumb for density, its categories evolved as we learn more about it, but we settled on boundaries on the extreme regions which gave us good explanatory power and resulted in categories of  $\{< 0.1, [0.1, 0.9], > 0.9\}$ .

Our next step was to populate this matrix with parameterized subject programs to achieve coverage of and balance across the cells. We began by utilizing the set of programs in Table 1 with their default parameter values. Since this initial set of programs did not even fill all cells in the matrix, we started manipulating those programs that provided input parameters to scale them up or down in order to cover the matrix. In some cases, the manipulations were quite simple. For example, many of the programs accept parameters to manipulate the thread count, which enabled us to scale them up in the thread dimension quite easily. Attempting to manipulate programs to cover the error density spectrum, however, often required studying the programs in detail to understand the nature of the fault and its relationship to program parameters. The process of subject parameterization and density measurement was repeated until we ended up with approximately five subjects in each cate-



**Figure 4: Interaction between error density and the number of threads.**

gory; given the non-uniform nature of the density categories there were more subjects in the middle. The resulting matrix of 56 parameterized subject programs is presented in Table 3. For each program, we provide values for the parameter settings defined in Table 1, within (...), and an associated pair of thread count and error density measures, within <...>.

We used the same execution platform and environment configuration as was used in the study described in the previous section.

### 4.2.3 Results and Analysis

We start exploring our conjecture through the analysis of the three graphs in Figure 4. Each graph depicts a dependent variable in the y-axis, the x-axis describes the three categories of thread count

and the lines representing the three categories of error density. Each observation in the graph corresponds to the mean value of the dependent variable across all programs in the cell corresponding to that thread and density category. The whiskers around the mean represent the 95% confidence interval.

The top graph shows that high density subjects tend to cover a relatively small number of new states independently of the number of program's threads. At lower densities, the picture is less clear. Closer inspection of the data revealed that analysis of a single program (RW) in the medium category has an enormous number of new states and raises the average dramatically. This is also confirmed by the large variation in the number of states covered by programs with a medium number of threads as evidence by the large variation observed around the means. Still, when error density is high, the number of new states explored is consistently small.

The middle graph on error depth shows a more consistent tendency. Only programs with low error density seem to provide larger error depths. Interestingly, on average, programs with a larger number of threads did not end up with deeper errors. The bottom graph presents a similar story. Programs with a higher error density tend to have easy to find faults, independent of the number of threads, while programs with lower error densities have harder to find faults. However, the number of threads does seem to affect whether the error is found or not, and it does compound with error density.

These preliminary findings clearly support one aspect of our conjecture, that is, error density is an important factor influencing the cost of path-sensitive error detection techniques. When these techniques check programs with high error density, they will find the errors and do so quickly. On the other hand, surprisingly, we did not find strong evidence for the influence of thread count on analysis cost. Programs with a high number of threads impacted cost only when density was medium or low, which indicates that thread count cannot be considered a strong factor in isolation.

It is important to observe, that error density is not a property of source code. Parameters can be chosen for some examples, e.g., BoundedBuffer, that have very low (.027) and very high (.996) error density. Researchers must take great care in selecting subjects and in qualifying research results in light of the degree of error density variation possible through program parameterization and its implications on the cost of analysis.

An interesting follow up conjecture is whether thread count and error density could be used to predict the number of new states explored, the error depth, or whether an error may be found or not. To explore such a conjecture and build effective predictive models we must collect and prepare a larger and more homogeneous set of programs that overcomes some of the limitations of the current pool. Note, for example, that not all our subject programs had parameters for scaling, and not one program could be scaled to fit in all the cells of Table 3, both of which would have been desirable to control the sources of variation due to particular program characteristics, and to build stable prediction models. This task is a difficult one considering that we have just begun to understand the size and structure of concurrent program state spaces and the effect they have on analysis cost.

## 5. RESULTS IN CONTEXT AND RECOMMENDATIONS

### 5.1 Threats to Validity

Empirical studies are subject to threats to validity; these threats must be considered in order to determine the soundness and significance of the results. We detail the threats to our studies and the steps we took to mitigate their impact on our findings.

**Internal Validity.** We placed an upper-bound on the execution time and memory that could be used by JPF during any analysis run. The bounds we chose were large, one hour and 4 GBytes respectively, and consistent with settings used in other studies. Changing those bounds may impact the findings on error detection, for instance, unlimited space and time would allow all errors to be found. During the course of our studies several defects were discovered and reported to the JPF team; fixes were provided and all studies were repeated on the updated version of JPF. We know of no defects remaining in JPF 3.1.2 that would affect the results of our study.

**External Validity.** Our studies consider version 3.1.2 of JPF only; different versions of JPF and different path-sensitive analysis frameworks may yield different results. Replicated studies with other tools would address this threat. The subjects chosen for this study were selected from a variety of sources. While the main goal of selecting subjects was to create a diverse set of multi-threaded Java programs containing safety property violations, we had two additional criteria: (1) selecting subjects that had been used in at least one previously published study in support of the studies in Section 3, and (2) selecting subjects that are either in widespread use, or are proposed as benchmarks, for evaluating path-sensitive analysis techniques in support of the studies in Section 4. Although we do not know if these programs are truly representative of multi-threaded Java programs in general, we believe our selection of programs from this initial population provides meaningful information with regard to our studies.

**Construct Validity.** The measures we considered in our studies are not the only possible measures of the influence of variation in search order and program factors on analysis cost. System-independent measures such as the number of seen (visited) states, the number of end states, the number of transitions, etc. could also have been used; however, the number of new states and the error depth are widely used in evaluating state-oriented analysis techniques. System-dependent measures such as memory usage and CPU time were not considered to be valid measures because they could skew the results for a given subject even within an isolated environment due to factors such as execution time spent on garbage collection. Furthermore, execution time for path-sensitive analyses is strongly dependent on the number of new states; an early version of our studies that used execution time bore this out, but did not allow for retrospective comparisons.

**Conclusion Validity.** One of the major lessons we learned through our studies is that the values of the dependent variables are not only large, but also extremely variable. As a result, the 1000 runs of JPF on each subject may have not been enough to appropriately characterize the error density of some programs. Our choice of the number of runs was incremental, increasing the number of runs until the observed standard variation seemed to stabilize, and it was also limited by data collection costs. This is one of the reasons we adopted primarily an exploratory rather than a formal analysis, where we try to characterize relationships rather than claim any type of causality between the independent and dependent variables.

## 5.2 Exposing and Controlling Search Order

Researchers wishing to evaluate the benefits of techniques that reduce the cost of path-sensitive error detection analyses should control for search order. We believe that it is not cost-effective for researchers to perform studies of the scale reported in this paper; our studies took multiple person-months and CPU-weeks to setup and conduct. We propose instead that developers of path-sensitive analysis tools *provide the ability to configure the default search order*. Tool frameworks such as JPF and Bogor make this relatively

easy, but for other tools, such as SPIN, this would require significant effort. With this ability, cross-tool studies would be able to ensure that tools use the same default order and intra-tool studies of new techniques will be able to evaluate the extent to which a technique's benefit is independent of default order. Even without the ability to control default search order, tool developers should, at a minimum, clearly describe the default search order implemented in their tool so that researchers using the tool will be able to properly qualify research findings based on the use of the tool.

## 5.3 Building Better Benchmarks

Our studies clearly demonstrate that programs with high path error density are poor subjects for evaluating error detection techniques. Fortunately, the community is beginning to move away from using concurrency kernels, which almost uniformly have high density, in evaluations.

Researchers have proposed a number of criteria for constructing benchmarks of programs for evaluating multi-threaded program validation tools. For example, simple criteria such as program size or the presence of language constructs (e.g., wait and notify) are sensible ways to build diversity in a benchmark. More insightful selection criteria involve varying thread counts and the types of errors in the benchmark (e.g., the IBM benchmark). The results of our study suggest that additional factors can significantly effect the difficulty of finding an error in a program. Path error density is one such factor, but the variation in Figure 4 suggests that there are others.

The community needs good benchmarks and to build good benchmarks we need to understand the variations in programs to which different validation techniques are sensitive in terms of cost and effectiveness. Clearly, more work is needed to achieve this, but we plan to continue the work on benchmark development [11, 16] by sharing all of the subjects in this study through the Subject Infrastructure Repository [4].

## 5.4 Future Directions

We believe that the studies reported in this paper provide a wealth of data that can be leveraged for future work. For example, we plan to explore the influence of path error density on other techniques for validating and testing multi-threaded programs to understand their sensitivity to that factor. We also believe path error density is just one measure that can be used to characterize an analysis problem. It has value because it is efficient to calculate, i.e., a small number of randomized simulations can indicate a program's path error density, and it appears to be useful in predicting when path-sensitive techniques will have an advantage over simpler techniques, such as randomized testing [25] for a given program. There may well be other factors that share these advantages.

In conducting our studies, we performed a very large number of randomized depth-first search runs using JPF. Data from these runs suggest that a cost-effective strategy for finding errors in systems with low path error density is to run multiple parallel randomized analyses, and then terminate all analysis runs when one of them finds an error. Since randomized analysis runs are completely independent this approach should scale well to very large degrees of parallelism.

## Acknowledgments

This work was supported in part by the National Science Foundation through awards 0429149 and 0444167 and through CAREER award 0347518 and by the Army Research Office through DURIP award W911NF-04-1-0104. We would like to thank Radu Iosif, Willem Visser, Peter Melhitz, Robby, and Matt Hoosier for dis-

cussions about search order in Spin, JPF and Bogor. We thank Todd Wallentine, John Hatcliff, and Venkatesh Prasad Ranganath for general discussions about transition order in Bogor. Finally, we thank Shmuel Ur and Yaniv Eytani for granting access and supporting our use of their benchmarks.

## 6. REFERENCES

- [1] J. C. Corbett. Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering*, 22(3), Mar. 1996.
- [2] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera Specification Language. *International Journal on Software Tools for Technology Transfer*, 2002.
- [3] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design*, October 1992.
- [4] H. Do, S. G. Elbaum, and G. Rothermel. Subject infrastructure repository. <http://esquared.unl.edu/sir>.
- [5] Y. Dong, X. Du, G. J. Holzmann, and S. A. Smolka. Fighting livelock in the gnu i-protocol: a case study in explicit-state model checking. *Int'l. Journal on Software Tools for Tech. Transfer*, 4(4):505–528, 2003.
- [6] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, Robby, and T. Wallentine. Evaluating the effectiveness of program slicing for model reduction of concurrent object-oriented programs. In *Proc. of the Twelfth Int'l. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, 2006. LNCS 3920.
- [7] M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting object escape and locking information in partial order reductions for concurrent object-oriented programs. *Formal Methods in System Designs*, 25(2–3):199–240, September–November 2004.
- [8] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Partial-order reduction and trail improvement in directed model checking. *International Journal on Software Tools for Technology Transfer*, 6(4), 2004.
- [9] <http://home.att.net/~ddavies/NewSmulator.html>.
- [10] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Toward a framework and benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice and Experience*, to appear.
- [11] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *Proc. of the Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2004.
- [12] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc. of the 17th Int'l. Symp. on Parallel and Distributed Processing*, 2003.
- [13] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th International Conference on Computer Aided Verification*, June 1997.
- [14] A. Groce and W. Visser. Heuristics for model checking java programs. *Int'l. Journal on Software Tools for Tech. Transfer*, 6(4):260–276, 2004.
- [15] D. Hamlet and J. Voas. Faults on its sleeve: amplifying software reliability testing. In *Proc. of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, pages 89–98, 1993.
- [16] K. Havelund, S. D. Stoller, and S. Ur. Benchmark and framework for encouraging research on multi-threaded testing. In *Proc. of the Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2003.
- [17] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
- [18] <http://www.kclee.de/clemens/java/javancss>.
- [19] M. Musuvathi and D. R. Engler. Model Checking Large Network Protocol Implementations. In *Proc. of the First Symp. on Networked Systems Design and Implementation*, Mar. 2004.
- [20] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proc. of the Fifth Symp. on Operating Systems Design and Implementation*, Dec. 2002.
- [21] C. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *Int'l. Journal on Software Tools for Tech. Transfer*, 5(1):34–48, 2003.
- [22] <http://research.microsoft.com/qadeer/cav-issta.htm>.
- [23] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.
- [24] Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic systems. In *Proceedings of the 2003 Workshop on Software Model Checking*, July 2003.
- [25] S. D. Stoller. Testing concurrent java programs using randomized scheduling. In *Proc. Workshop on Runtime Verification*, 2002.
- [26] J. Tan, G. S. Avrunin, L. A. Clarke, S. Zilberstein, and S. Leue. Heuristic-guided counterexample search in flavors. In *Proc. of the 12th ACM SIGSOFT Twelfth Int'l. Symp. on Foundations of Software Engineering*, pages 201–210, 2004.
- [27] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of the 15th IEEE Conference on Automated Software Engineering*, Sept. 2000.
- [28] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proc. of the Seventh Symp. on Operating Systems Design and Implementation*, Dec. 2004.