IEEE Copyright Notice

Published in: ***Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE'09),*** November 2009

"State-Space Coverage Estimation"

Cite as:

Ali Taleghani and Joanne M. Atlee. 2009. State-Space Coverage Estimation. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09). IEEE Computer Society, Washington, DC, USA, 459-467.

BibTex:

```
@inproceedings{Taleghani:2009:SCE:1747491.1747540,
 author = {Taleghani, Ali and Atlee, Joanne M.},
 title = {State-Space Coverage Estimation},
 booktitle = {Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering},
 series = {ASE '09},
 year = {2009},
 pages = {459--467}
}
```

DOI: https://doi.org/10.1109/ASE.2009.24

State-Space Coverage Estimation

Ali Taleghani
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
ataleghani@uwaterloo.ca

Joanne M. Atlee
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
jmatlee@uwaterloo.ca

***Abstract*—Software model checking is the process of systematically exploring a program's state space to find hard-to-discover errors. Because of the exponential size of the state space, an exhaustive search of the state space is often impossible given the memory resources. In such cases, an estimate of how much of the state space is covered can help the verifier to decide whether to employ additional computational resources or to use more aggressive abstraction techniques.**

Our work focuses on coverage estimation for explicit-state model checking of software programs. In this paper, we present an estimation algorithm that is based on Monte Carlo techniques that sample the unexplored portion of the reachability graph. We implemented our algorithm in Java Pathfinder and evaluated our approach on a suite of Java programs, simulating out-of-memory errors after a known percentage of a program's state space had been searched. Our empirical studies show that, on average, our algorithm's coverage estimates differ from the actual coverage by less than 10 percentage points, with a standard deviation of about 5 percentage points – regardless of whether the actual state-space coverage is low (3%) or high (95%).

I. Introduction

Model checking is a search-based verification technology that examines a model's or program's state space and verifies its conformance to desired properties. One of the main obstacles to wide-spread use of model checking is the *state explosion problem:* the size of a model's or program's state space grows exponentially with the number of variables and concurrent processes [6]. Many programs are too large to be checked exhaustively, even after using state-of-the-art state-space reduction techniques like partial-order reduction [12], predicate abstraction [13], or slicing [23].

If a model-checking search ends prematurely without achieving full coverage or finding an error, an estimate of the search's state-space coverage could be very useful. For example, the degree of coverage may offer some level of confidence in the partial results: finding no errors when searching 95% of the state space is more significant than finding no errors when searching 10% of the state space. Alternatively, the estimated coverage could suggest how best to improve the coverage on subsequent model-checking searches: high coverage suggests that running the model checking problem on a more powerful machine (with more memory) might be sufficient to achieve full coverage, whereas low coverage suggests that more aggressive abstractions or parallel model checking would be required. Of course, to be useful, the coverage estimation needs to be somewhat accurate.

In this paper, we propose an algorithm for estimating the state-space coverage of a model-checking search that terminates prematurely due to insufficient memory. Our algorithm uses Monte Carlo techniques to sample unexplored transitions in the reachability graph, count the number of unvisited states reachable via these transitions, and extrapolate from this an estimation of the number of states still unvisited when the search terminates.

Our contributions are as follows:

1) **Coverage estimation:** An algorithm for estimating the state-space coverage of a partial model- checking search. The algorithm design makes no assumptions about the associated model checker, so we expect that the algorithm can be embedded in any explicit-state model checker.
2) **Implementation:** A prototype embedding of our algorithm in the Java PathFinder model checker.
3) **Evaluation:** An empirical study of our algorithm on a suite of nine Java programs. The prototype was tuned using a subset of these programs, to improve the accuracy of its estimations. We then evaluated the algorithm and prototype using the full suite of programs. The search of each program was terminated prematurely, after reaching varying degrees of partial state-space coverage (ranging from 3% to 95% coverage), and the reported coverage estimate was compared against the actual coverage (which was known). Results show that, on average, our algorithm's coverage estimates differ from the actual coverage by 3 to 10 percentage points, with a standard deviation of about 5 percentage points.

This paper is organized as follows. Section II introduces our estimation algorithm. In Section III, we describe its implementation within the Java PathFinder model checker, and discuss our evaluation methodology and suite of test programs. The results of our evaluations are presented in Section IV. We discuss in Section V alternative approaches and optimizations that we have explored in the course of our research. We end with a discussion of related work in Section VI and conclusions in Section VII.

II. State-Space Coverage Estimation

Some programs are too large to be exhaustively model checked, in which case we would like the model checker to report the percentage of the state space covered during a

partial search. In these circumstances, the model checker has two goals: to estimate the percentage of state space covered by a search, and to explore and examine the program's state space. It may be that the best search strategy for the purpose of verification is not the best search strategy for the purpose of coverage estimation. In general, we would expect a verification search to be a systematic exploration of the entire state space, whereas an estimation search should cover different parts of the state space to provide an accurate estimation of state-space coverage.

Thus, we take a two-pronged approach to estimation. During the first phase (*verification phase*) we devote some percentage of memory to an exhaustive search. Thus, the model checker must be able to keep track of the amount of memory utilized as a percentage of total memory available. If the verification-phase memory limit is reached before model checking completes, then the model checker switches strategy and uses the remaining memory for estimation.

During the second phase (*estimation phase*), the model checker performs partial state-space searches starting from multiple unexplored transitions in the reachability graph. Figure 1 shows how a reachability graph might be covered by our estimation algorithm, with some states being visited during the algorithm's verification phase and other states being visited during the estimation phase.

Despite their names, each phase contributes to both verification and estimation: during the verification phase, a count of the number of unique states visited is maintained (to be used in calculating the estimated coverage); and during the estimation phase, newly discovered states are verified on discovery. Thus, even if we set aside some memory for the purpose of estimation, that memory will be used to explore more states. The estimation phase ends when either memory is exhausted or the state space is fully explored.

Our estimation algorithm uses Monte Carlo techniques to sample different parts of the unexplored state space and to extrapolate an estimate of the number of unexplored states. At any point in time, the search maintains (1) a hash table of visited states (in which distinct states are represented as unique integers called *fingerprints*), and (2) a *worklist* of partially explored states that have been reached and verified, but whose transitions are still to be fully investigated. During the verification phase, the worklist is used to co-ordinate a systematic search of the program's state space. During the estimation phase, the worklist serves as a source of unexplored transitions to be sampled. The algorithm searches a random selection of unexplored transitions and counts how many unvisited states are reachable from each transition.

We assume that the number of new states reachable from a set of sampled transitions tells us something about the number of unvisited states that would be reached from the unsampled transitions. That is, we assume that the ratio of (a) the number of new states discovered during the estimation phase to (b) the number of sampled transitions is comparable to the ratio of (c) the number of states that remain unvisited at the end of the estimation phase to (d) the number of transitions left unexplored at the end of the estimation phase:

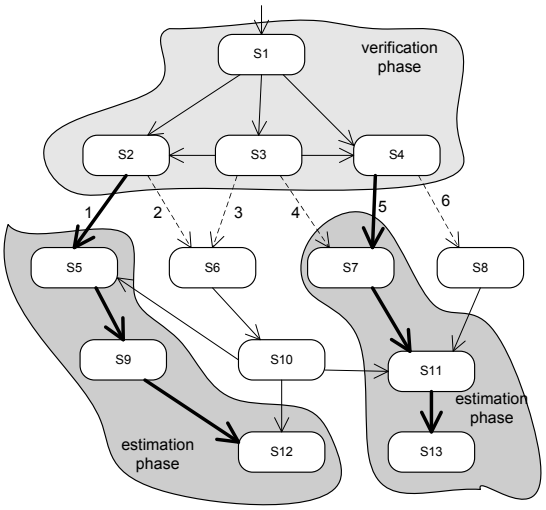

Fig. 1. Schematic example of our estimation algorithm

$$\frac{(a)\#states\ found\ during\ sampling}{(b)\#\ sampled\ transitions} \approx \frac{(\mathbf{c})\#\ \mathbf{unvisited\ states}}{(d)\#\ unexplored\ transitions} \quad (1)$$

The estimation algorithm measures the italicized values in Equation 1 and solves for the number of unvisited states.

During our investigation, we discovered that we obtain more accurate results if the estimation phase samples and counts only *productive* transitions, where a transition is *productive* if it leads to an unvisited state. By disregarding the transitions that lead to already visited states, we improve the estimated ratio of new states per sampled transition. The modified equation is as follows.

$$\frac{(a)\#states\ found\ during\ sampling}{(b2)\#\ sampled\ productive\ transitions} \approx \frac{(\mathbf{c})\#\ \mathbf{unvisited\ states}}{(d2)\#\ productive,\ unexplored\ transitions} \quad (2)$$

Note that whether a transition is productive changes during the course of the search: a transition that is deemed productive at the start of the estimation phase may later be deemed unproductive if in the meantime its destination state is visited. Thus, the count of productive transitions is more accurate if it is made at the end of the estimation phase.

In Figure 1, for example, the verification phase ends with three states in the worklist (s2, s3, s4) that together have six unexplored transitions emanating from them (numbered, for pedagogical purposes only, 1 to 6). Suppose further that, during the estimation phase, the model checker samples *two* transitions, 1 and 5, and discovers a total of *six* new states before it runs out of memory. At the end of the estimation phase, four transitions remain unexplored (dashed transitions), of which only *two* transitions are productive and lead to

unique destination states. Using these values in Equation 2, the estimated number of unvisited states is $(6 \div 2) \times 2 = 6$.

It is important to mention that by considering productive transitions only, our algorithm deviates from traditional Monte Carlo techniques. Normally, sampling is performed on the full data set and it is assumed that the data set does not change as a result of sampling. In our case, however, the data set (unexplored productive transitions in the worklist) changes throughout the estimation phase because the exploration of a sampled transition may cause other transitions in the worklist to become unproductive. As a result, the number of productive transitions that remain in the worklist at the end of sampling might be an overestimate, since not all transitions would be deemed productive if the sampling were exhaustive. Still, counting the number of productive transitions that remain in the worklist at the end the estimation phase results in a smaller overestimation of productive transitions than if the transitions were counted at the start of sampling.

Once we obtain the estimated number of unvisited states, we compute the estimated coverage using Equation 3. $UnVisited$ is the estimated size (number of states) of the unexplored state space, obtained from Equation 2, and $Visited$ is the number of unique states discovered during the combination of the verification and estimation phases:

$$\%Coverage = \frac{Visited}{Visited + UnVisited} * 100 \qquad (3)$$

Thus, to complete the example shown in Figure 1, the estimated coverage would be $(10) \div (10+6)$ = 63%, when in fact the verification and estimation phases together cover 77% of the state space.

In the next sections, we describe the verification and estimation phases in more detail.

A. Verification Phase

The main purpose of the verification phase is to verify the program and discover any property violations. If the verification phase ends without covering the entire state space, we want a large worklist to sample from during the estimation phase. Larger worklists usually lead to more accurate estimation results.

Our algorithm employs breath-first search (BFS) for this phase. BFS is less efficient than a depth-first search (DFS) because there is more context switching between states. BFS tends also to be less memory efficient than DFS because its worklist (a queue) contains all states at the current level of the search, where *level* is the path length from the initial state in the reachability graph; in contrast, the worklist of a DFS (a stack) contains only the states in the path from the initial state to the state currently being explored. That said, a large worklist is advantageous for sampling and estimation, so we use BFS in the verification phase despite its inefficiencies.

Another advantage of using BFS during the verification phase is that it ensures an exhaustive search of execution paths up to some length determined by the verification-phase memory limit – which is, effectively, a form of bounded model checking [4].

B. Estimation Phase

The goal of the estimation phase is to (1) estimate the ratio of unvisited states per unexplored transition (left-hand side of Equation 2), and (2) count the number of productive transitions that remain unexplored at the end of the estimation phase (value of (d2) on the right-hand side of Equation 2). We describe how to obtain both values below.

1) Number of Unvisited States per Unexplored, Productive Transition: We sample a subset of the unexplored transitions emanating from the states in the worklist, and count the number of unvisited states that are reachable from each transition. If a sampled transition leads to an already-visited state, then it is deemed *unproductive* and we pick another transition.

Each sample is an exhaustive search of the state space reachable from some productive transition. Either BFS or DFS could be used. We chose to use DFS because it is faster and more memory efficient, as explained above.

To obtain accurate estimation results, it is desirable to sample the reachability graph as uniformly as possible. Thus, to improve the breadth of the estimation search, the algorithm randomly selects unexplored transitions from the worklist.

The algorithm continues to sample transitions until either no more unexplored transitions remain in the worklist or the memory allocated to the estimation phase is exceeded. The former case corresponds to an exhaustive search of the state space and no estimation is necessary. In the latter case, we calculate the average number of unvisited states that each sampled, productive transition discovers.

2) Number of Remaining Unexplored Productive Transitions: At the end of the estimation phase, we count the number of unexplored productive transitions that remain in the worklist. For that, the model checker traverses the worklist, executes every unexplored transition of every state in the list, and checks whether the destination state is unvisited. The model checker does not explore beyond the destination state. The result is the total number of productive transitions that remain unexplored.

During this final count of productive transitions, the model checker discards destination states as they are created. It does, however, continue to store the fingerprints of the destination states in a hash table of visited states, in order to recognize repeat visits to the same state. Thus, this step requires negligible additional memory.

C. Memory Management

The verification phase and estimation phase both utilize memory: in both phases, the model checker stores partially processed states in a worklist and separately maintains fingerprints of visited states in a hash table. How we divide the available memory between the two phases can affect the accuracy of the estimation results.

In general, we would expect to obtain more accurate coverage estimations if the verification phase reaches deeper into

the state space before the estimation phase starts. This is because the shape of the reachability graph may not be regular and may contain bottlenecks or regions that are reachable via only a few transitions. If the verification phase is thorough enough to progress through these bottlenecks, then the unexplored portions of the reachability graph that remain are more strongly connected and are more equally reachable via a random sampling.

On the other hand, when the size of available memory is very small compared to the size of memory needed for an exhaustive search, it is important that there be enough memory available during the estimation phase so that the estimation searches reach and count a sufficiently large number of new states per sampled transition. In these cases, allocating more memory to the estimation phase is more effective. It is important to recall that, no matter how we split memory between the two phases, *all* available memory is used to search the state space (and check for property violations) because even during the estimation phase the model checker is still examining new states.

We experimented with different ratios of memory allocated to each phase and report the results in Section IV.

III. Evaluation Method

We evaluated our coverage estimation algorithm by embedding it into Java Pathfinder (JPF) version 4 [15]. JPF is a software model checker that verifies a Java program by exhaustively exploring the program's executable state space and looking for program states that result in deadlocks or that violate user-specified assertions.

As evaluation programs, we selected programs with known state-space sizes because this allows us to evaluate how well our estimation algorithm performs. We evaluated our work on a suite of nine Java programs that come from five sources and that have been used in previous empirical studies. The programs are relatively diverse with respect to their state-space size, the ratio of transition to states, the number of parallel threads in each program and the diversity of the concurrent processes in each program (e.g. the dining philosopher program has identical processes, whereas the sleeping barber program consists of several different processes). Table I lists each program including a citation of its source, the program parameter values that we used (e.g., we instantiated eight dining philosophers), and the resulting size of the program's reachability graph in terms of numbers of states and transitions. We used the first four programs of our evaluation suite as *tuning programs* to fine-tune the memory allocation of our algorithm, and we used the remaining five programs (shown in dark rows) for evaluation only.

To simulate out-of-memory scenarios, we vary the percentage of the programs' total state space that the model checker searches. Specifically, we prematurely terminate the model checker once 3%, 10%, 25%, 50%, 75% and 95% of the program's state space has been covered. We refer to these six memory thresholds as *coverage limits*. Within this coverage limit, our algorithm performs verification- and estimation-phase searching and calculates a coverage estimate. We then compare the estimated coverage with the known state-space coverage (i.e., the coverage limit). We used coverage limits 10%, 25%, and 75% (referred to as *tuning limits*) to fine-tune our algorithm and determine how much memory to allocate to the verification phase versus the estimation phase. We used the remaining limits solely for evaluation. In practice, when the size of a program's state-space is unknown, we use JPF's facilities for keeping track of memory usage to determine when the verification phase has used up all of its allocated memory (a fixed percentage of the total available memory). Thus, our approach could be easily adapted to any explicit-state model checker that supports both DFS and BFS, and includes facilities for tracking memory usage during the search.

We ran our experiments on an Intel Pentium 4 3.2GHz machine with 1.5GB of memory, running Windows XP.

IV. Experiments and Results

In this section, we present the experiments for evaluating our estimation algorithm and the results we obtained.

In the first set of experiments, we varied the amount of memory allocated to each phase of the algorithm, and we compared the accuracies of the resulting estimations. In particular, we varied the amount of memory allocated to the verification phase to range between 40% and 90% of available memory (artificially restricted by the coverage limit, as described in the previous section), in 10% increments. The rest of the memory (minus a small amount to compute the estimation at the end) is allocated to the estimation phase. We performed this experiment for all tuning programs and tuning coverage limits.

The results showed that for low coverage limits, where a search terminates before a significant fraction of the state space is explored (10% to 25%), it is best to allocate 50% of available memory to the verification phase. For higher coverage limits (75% and higher), it is best to allocate 70% of available memory to the verification phase. In practice, though, one does not know ahead of time whether the model checker is likely to cover a small or large fraction, or all, of a program's state space. For all of our subsequent experiments, we used an allocation that lies in the middle of the above two values: 60% of available memory in the verification phase, for all coverage limits. More extensive experimentation is needed, involving a very large and diverse set of programs with known state-space sizes, to determine a more precisely optimal allocation of memory.

Our second set of experiments evaluate how well our estimation algorithm works for all programs and all coverage limits in our evaluation suite. We model checked each program with respect to each coverage limit ten times and report the results in Table II. The *deviation* between a coverage estimate and a search's actual coverage (set by the coverage limit) is expressed in terms of percentage points: the absolute value of the difference between the two percentages of state space.

TABLE I
JAVA PROGRAMS USED FOR EVALUATION

	Source	Program	Parameters	States (S)	Transitions (T)
1	[15]	Dining Philosopher (8)	#philosophers	209014	1093394
2	[3]	Bounded Buffer (5,4,4)	buffersize,#prod,#cons	786987	5606270
3	[2]	Nasa KSU Pipeline (3)	stagesize	59512	246697
4	[3]	Nested Monitor (5,4,4)	buffersize,#prod,#cons	71941	493776
5	[3]	Pipeline (7)	stagesize	82011	500823
6	[3]	RWVSN (4,4)	#readers,#writers	227116	1167826
7	[3]	Replicated Workers (5,2)	#workers,#items	710022	3648744
8	[8]	Sleeping Barber (2,4,3)	#barber,#customers,#chairs	1452194	6295518
9	[1]	Elevator(5,10,10)	#elevators,#floors,#people	386032	3232346

We report the smallest deviation (column *Best*), the largest deviation (column *Worst*), the average deviation of ten runs (column *Avg*), and the standard deviation of the deviations (column σ). For example, consider a search of the pipeline program with a coverage limit 25%. A perfect estimate would report that 25% of the program's state space had been covered by the search. The best estimate (out of ten) reported by our algorithm was off by 2 percentage points, the worst estimate was off by 18 percentage points, the average deviation was 10 percentage points; and the standard deviation from the average estimate was 7 percentage points. Considering all nine programs and six coverage limits, with each combination run ten times, our worst coverage estimate was off by 37 percentage points. We recorded whether the reported coverages were overestimatations or underestimations, but we could not detect a clear bias. It is for this reason that we report deviations as absolute values.

The standard deviation illustrates the variability of our results: one *standard deviation* specifies the range of values, centred around an average, within which 60%-70% of estimates fall, assuming a normal distribution. Thus, a standard deviation of 5 percentage points indicates that most of our estimates fall within 5 percentage points of the average estimate. We believe the variability of the results is primarily due to different reachability graph properties of our evaluation suite. We discuss in Section V factors that could affect the accuracy of the results.

To evaluate the performance overhead of our estimation algorithm, we compared its execution time to that of traditional model checking. To eliminate variability due to the search strategy, both our algorithm and the traditional model checking runs use BFS to search 60% of each program's state space and use DFS to search the remaining 40% of the state space. The results showed negligible difference in performance. This is not surprising, given that the only extra work that our algorithm performs is the counting of transitions, which is negligible compared to the search and verification of the state space. To assess the performance overhead of using BFS during the verification phase, we also compared our algorithm's execution time to that of a DFS model checking search. The results showed that, depending on the coverage limit, the overhead ranges between 12% and 38%.

V. DISCUSSION

Throughout our work, we experimented with various estimation techniques and optimizations of our current algorithm. In this section, we describe lessons learned with respect to the most important experiments.

A. Rate of Discovering New States

It seems intuitive that the rate of discovering new states would decrease during the course of a search and that we can use this information to improve our coverage estimate. In particular, the model checker could keep a running ratio of transitions to states, and could compare the current rate of new-state discoveries (measured at fixed intervals) against the overall ratio.

To test this hypothesis, we performed exhaustive searches of our tuning programs and counted, at fixed intervals, the fraction of transitions that are productive (i.e., that lead to new states). Figure 2 shows the rate of discovering new states for one of our evaluation programs. The x-axis shows the progress of the search in terms of the percentage of all transitions explored.

As can be seen, the rate of discovering new states drops quickly at the start of the search and then decreases slowly for the rest of the search, maintaining a surprisingly moderate rate even at the end of the search. All evaluation programs exhibit similarly shaped graphs, although the steep drop occurs at different stages of the search for different programs. Given the comparable rates throughout most of a search, including up to the end of a search, we were not able to deduce any particular properties that could be used to improve coverage estimation.

Given that the estimation phase of our algorithm measures, in effect, the rate of newly discovered states, it is conceivable that our algorithm overestimates coverage if sampling occurs early in the search when the rate of discovering new states is much higher (e.g. when the steep drop occurs in Figure 2). For our experiments, we expect that parts of the estimation phase would occur during this stage of higher rate of discovering new states, especially at the coverage limits of 3% and 10%,. However, we did not observe any overestimations that were surprising compared to other experimental results. It is possible that this sharp drop actually helps to obtain more accurate results when coverage is low (e.g., to counterbalance

TABLE II
STATE-SPACE COVERAGE ESTIMATION RESULTS

	Coverage Limit	3%				10%				25%			
	Estimate Deviation (% points)	Best	Worst	Avg	σ	Best	Worst	Avg	σ	Best	Worst	Avg	σ
Program	Dining Philosophers	4	14	7	4	3	11	8	4	4	16	9	5
	Bounded Buffer	3	5	3	2	1	4	2	3	3	16	9	6
	Nasa KSU Pipeline	2	2	1	2	2	7	4	3	4	12	7	3
	Nested Monitor	2	4	2	2	2	10	6	4	3	13	7	5
	Pipeline	4	10	8	3	1	11	5	3	2	18	10	7
	RWVSN	1	2	1	1	1	7	5	3	2	6	2	3
	Replicated Workers	9	25	16	7	3	9	6	4	2	10	3	4
	Sleeping Barber	2	7	5	2	2	11	5	4	3	10	6	3
	Elevator	1	2	1	2	2	7	4	2	2	9	2	5
	Average	3	8	5	3	2	9	5	3	3	12	6	5

	Coverage Limit	50%				75%				95%			
	Estimate Deviation (% points)	Best	Worst	Avg	σ	Best	Worst	Avg	σ	Best	Worst	Avg	σ
Program	Dining Philosophers	5	15	10	4	2	23	15	10	5	19	10	5
	Bounded Buffer	7	19	11	5	12	31	23	7	4	37	14	13
	Nasa KSU Pipeline	1	5	3	2	3	18	6	7	5	15	6	4
	Nested Monitor	2	9	5	4	5	20	13	3	6	14	10	3
	Pipeline	2	9	6	3	3	14	5	5	4	8	4	2
	RWVSN	7	21	18	6	2	16	10	6	4	15	10	4
	Replicated Workers	10	23	8	7	7	17	8	5	6	16	12	4
	Sleeping Barber	8	24	15	7	3	14	5	4	2	8	5	3
	Elevator	1	9	5	4	11	27	7	9	2	5	2	2
	Average	5	15	9	5	5	20	10	6	4	15	8	4

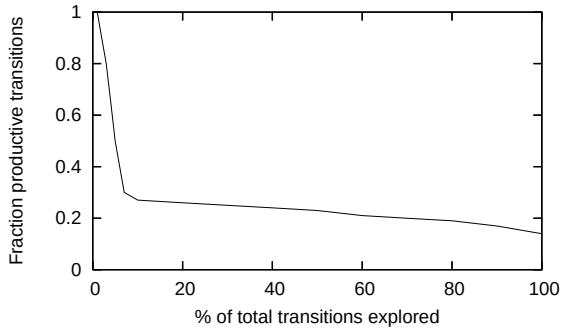

Fig. 2. Rate of discovering new states for the dining philosopher program

underestimations that are due to unexplored bottlenecks in the reachability graph) or it is possible that the drop is so steep that it occurs for only a very small portion of the estimation phase, and thus the estimation results are largely unaffected. In either case, more experiments with a larger set of evaluation programs are needed to determine the significance of this possible overestimation.

B. BFS Level Graphs for Estimation

We might expect that a breadth-first search of a program's state space would produce a worklist whose size varies regularly and predictably over the course of a complete model checking run. That is, in early phases of the search, the size of the worklist grows and during later phases of the search, the size of the worklist shrinks.

The authors of [7], [18] assume that the size of the worklist, measured after searching each level of the reachability graph, has a regular distribution. In this work, a *BFS level* of k is the set of states that are reachable from the initial state via a path of length k. In [18], authors plot the number of unprocessed states that are in the worklist at each BFS level and showed partial BFS level graphs to human subjects, who tried to guess the shape of the full graph. Given the results from the human experiments, the authors then deduced some parameters that they used to estimate state-space coverage based on the predicted shape of a search's BFS level graph. The authors of [7] use least-squares fitting of partial BFS level graphs to estimate the total number of states.

Our own experiments, however, indicate that the size of a BFS worklist does not necessarily have a regular or predictable distribution and thus may not be a reliable basis for coverage estimatation. Figures 3 and 4, for example, show the BFS level graphs for the elevator and RWVSN programs, respectively. Neither of these graphs have regular or bell-shaped curves. In our evaluation suite, six programs had a relatively normal distribution and three did not.

C. BFS vs. DFS During the Verification Phase

In our approach, an important design alternative is the search strategy used during the verification phase. DFS is popular because it is fast: the program stack can be used to store the worklist of partially explored states, so there is less context switching when the next state is explored; and the size of the worklist is relatively small since it contains only states along a single execution path. However, we use BFS because we hypothesize that having a larger worklist at the start of the estimation phase results in a more accurate estimation.

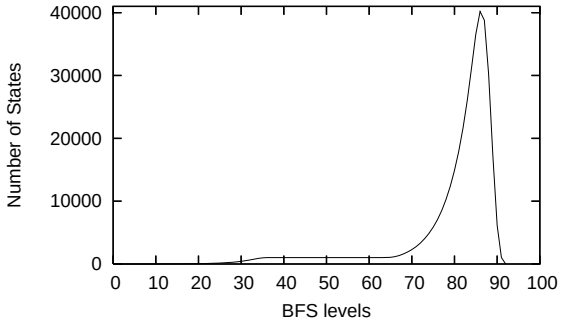

Fig. 3. BFS level graph for Elevator program

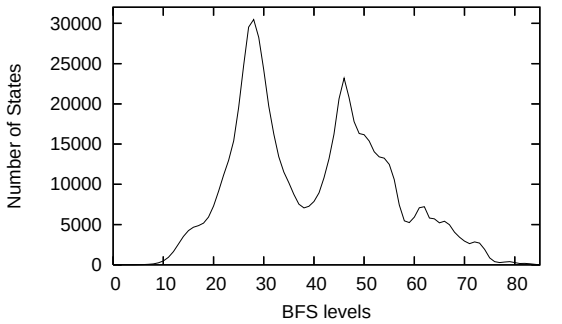

Fig. 4. BFS level graph for RWVSN program

To test this hypothesis, we experimented with using DFS rather than BFS during the verification phase. In a study of nine evaluation programs and six coverage limits (54 cases), using DFS produced less accurate results in 44 cases by an average of 14 percentage points; produced equally accurate results in 3 cases with an average difference of 1 percentage point; and produced more accurate results in 7 cases by an average of 7 percentage points. The results confirm that using BFS during the verification phase is likely to improve the accuracy of our algorithm's coverage estimates.

D. Round-Robin Execution of Estimation Phase Searches

One risk of the current design for the estimation phase of our algorithm is that the remaining memory is exhausted while searching the first sampled (unexplored) transition, and that this can result in estimates that are wildly off-base: the estimation may be way too high if the number of new states that are reachable from this one transition is much higher than the actual average number of new states per unexplored transition. We hypothesized that we could improve the accuracy of our estimations by sampling multiple unexplored transitions at once, in round-robin fashion.

To test this hypothesis, we modified the estimation phase of our prototype to sample several unexplored transitions in parallel in a round-robin fashion. The model checker keeps a separate DFS stack (worklist) for each sampled transition, and stores partially processed states for each DFS in that DFS's local worklist. There is one shared global hash table that stores fingerprints of visited states. If a DFS finishes before memory is exceeded, then the model checker picks a new unexplored transition from the worklist and starts a new DFS.

To evaluate this technique, we varied the number of transitions that are sampled in parallel and evaluated the accuracy of the resulting estimation. We observed that when the estimation algorithm samples five to ten unexplored transitions in parallel, the accuracy of its coverage estimate improves for (tuning) coverage limits of 10% and 25% but worsens for the coverage limit of 75%. When the number of parallel searches is above 15, then estimation accuracy improves for the coverage limit of 75% but worsens for the coverage limits of 10% and 25%.

It seems that when state-space coverage is low, it is better to sample a smaller number of transitions so that the searches of the sampled transitions finish. If too many transitions are sampled, then the average number of new states per sampled transition is low and the algorithm underestimates coverage. The opposite is true when state-space coverage is high.

In general, however, we do not know in advance whether state-space coverage will be low, high, or complete, and thus we do not know how many transitions to sample. This method may become more applicable if there were a way to determine on the fly whether the coverage is likely to be low or high. For example, we are exploring the possibility of performing the estimation phase of our algorithm more than once, in which case the estimated coverage from one execution could be used to tune the estimation algorithm in the second estimation phase.

E. Scalability

Even though our experiments were performed on relatively small evaluation programs, we do not believe that larger programs would necessarily better demonstrate the effectiveness of our algorithm. This is because our algorithm depends more on different state-space properties rather than the size of the state space. For example, the number of bottlenecks in the state space (i.e. states that have very few incoming transitions, but can reach many other states) can have a large effect on the accuracy of estimation. We expect that the accuracy of estimation is lower for state spaces that have many bottlenecks. Also, if the rate of discovering new states and the fraction of transitions that are productive vary for different parts of the state space, then we expect the results to be less accurate. Thus, while studies on larger programs can be useful, we believe that testing our algorithm on a more diverse set of state spaces, regardless of their size, could offer more conclusive results.

VI. Related Work

Our work is most closely related to research on coverage estimation and is somewhat related to work on randomized searching and partial searches. We describe each area below.

A. Coverage Estimation

In previous work [22], we suggested that it may be possible to sample unexplored transitions as a means to estimate the size of the state space, but we did not explore this idea further. Other works related to state-space coverage estimation

are [7], [18]. The authors in [18] describe two techniques for estimating state-space coverage. In the first technique, they execute two random partial searches of the state space and use the overlap between the two searches to estimate coverage. In the second method, they use partial BFS level graphs as explained in Section V. The authors evaluate the accuracy of their coverage estimation algorithm based on whether it can classify the actual coverage of a search into the correct coverage range: $<$ 3%, 4%-25%, or 26%-100%. They evaluated their algorithm on 160 reachability graphs. In the best case, their algorithm is correct 72% of the time, and in the worst case it is correct 42% of the time. If we use the same three coverage ranges to evaluate the accuracy of our estimation algorithm, then our algorithm estimates the correct range 93% of the time in the best case, and 78% of the time in the worst case.

The authors in [7] try to estimate the actual size of the state space by applying least-squares fitting to partial BFS level graphs. As described before, the main assumption of this work is that BFS level graphs have regular, bell-shape curves that can be described by a quadratic formula: $y = ax^2 + bx$ for some a and b. The authors report the results for three coverage limits (25%, 50% and 75%) and three evaluation programs. Their results show that their algorithm can estimate the size of the state space with a 66% to 93% accuracy. As mentioned before, our results show that the BFS-level graphs of programs are not predictably regular.

Others have explored how a state-space search relates to code coverage [10], [14] or to specification coverage [10]. Rodrguez et al. [10] describe and implement a framework for the Bogor model checker [19] that supports branch coverage and specification coverage. Branch coverage judges how many of the branches in the code have been exercised, in cases where the program's environment can affect whether all of the program's execution paths are examined. Specification coverage indicates how much of a JML specification is exercised when model checking the program against the specification, given a particular program environment. Gore et al. [14] describe a branch coverage module for JPF that tries to exercise all branches of a program and reports the percentage of branches covered. None of these works estimate state-space coverage.

B. Random and partial state-space searches

Our estimation phase could be viewed as a random search of the state space, in that the sampling is a collection of random, partial searches of unexplored regions of the reachability graph. In general, random and partial state-space searches are often employed when the state space is too large for exhaustive model checking; the goal is to find errors rather than to prove their absence.

The idea of random walks and randomized state-space search was first suggested by West [24]. In each step of a random walk, the algorithm randomly chooses a successor of the current state and visits it. If the current state does not have any successors, the algorithm restarts from the initial state. Since the original random walk method was introduced, many optimizations have been suggested to improve its effectiveness at finding errors. These optimizations include re-initializing the search frequently to avoid getting trapped in strongly connected components [17], performing local exhaustive searches once a certain depth has been reached [21], keeping a small cache of visited states [11], and running parallel random walks [11], [21].

Randomly selecting from states and transitions in the worklist is explored in [9], [16], [20]. The Lurch model checker [16] uses random walks to perform partial searches of large state spaces. When exploring more than one path, Lurch inserts newly discovered states at random indices in the worklist to randomize the search. Dwyer et al. [9] perform random searches of the state space by randomizing the order in which child states are explored. They parallelize this method by distributing the search to multiple non-communicating machines. Rungta and Mercer [20] provide a randomized algorithm for directed model checking that randomizes the order of states in the priority queue that have the same heuristic value. This algorithm allows the distribution of the search to multiple machines.

Finally, bounded model checking [4], [5] is a form of partial search that is exhaustive up to some bound k on the length of execution traces. If no bug is found, one increases k until either a bug is found or some pre-defined upper-bound is reached. The verification phase of our algorithm, which is an initial BFS of the state space, can be viewed as a bounded search whose bound is determined by the amount of memory allocated to the search phase.

VII. Conclusion

We have presented a strategy for estimating the state-space coverage of a model checking search that terminates prematurely due to insufficient memory. We have implemented our algorithm in Java Pathfinder and have evaluated the implementation on a suite of Java programs.

In the future, we would like to explore possibilities for improving the efficiency and accuracy of coverage estimation. The performance overhead reported in Section IV is a direct consequence of our decision to use bread-first-search during the verification phase of our algorithm. It is worth investigating whether other search strategies or a mixture of strategies would more efficiently achieve our goal of accumulating a reasonably sized worklist. As an example alternative, the verification phase could be a depth-first examination of the program's state space, and the estimation phase could start by performing short breadth-first searches of states in the worklist, to increase the size of the worklist before estimation searches begin.

To improve accuracy, we are exploring strategies that employ multiple estimation runs, such as merging the results from independent estimations or incrementally refining an estimation. We also plan to investigate other state-space properties that could serve as preliminary indicators of coverage. Such indicators could be used to tune the estimation algorithm on the fly (e.g., tuning the percentage of memory allocated to the verification phase versus the estimation phase based on

early indications as to whether the state-space coverage will be low or high). Another interesting extension of this work would be to investigate techniques for reporting a bound on the uncertainty of the coverage estimation. An uncertainty bound would act as a measure of confidence in the coverage results and could be used to decide how to improve the coverage on subsequent searches. If the reported uncertainty bound is low, then the human verifier is more likely to trust the coverage results and adjust her verification technique based on the results. For example, a low coverage estimate with low uncertainty might suggest that she use more aggressive abstraction techniques.

REFERENCES

[1] http://home.att.net/ ddavies/newsmulator.html.

[2] http://verify.stanford.edu/uli/icse/workshop.html.

[3] http://www.cis.ksu.edu/santos/case-studies/counterexample_case_study.

[4] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of the Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[7] N. J. Dingle and W. J. Knottenbelt. State-space size estimation by least-squares fitting. In *Proceedings of the 24th UK Performance Engineering Workshop*, page 347357, 2008.

[8] M. B. Dwyer et al. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Proc. of the Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 73–89, 2006.

[9] M. B. Dwyer et al. Parallel randomized state-space search. In *Proc. of the 29th Int. Conf. on Software Eng.*, pages 3–12, 2007.

[10] Edwin Rodríguez et al. A flexible framework for the estimation of coverage metrics in explicit state software model checking. In *Proc. of the 2004 Int. Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004.

[11] Enrio Tronci et al. A probabilistic approach to automatic verification of concurrent systems. In *Proc. of the Asia-Pacific on Software Eng. Conf.*, page 317, 2001.

[12] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[13] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Proc. of the Int. Conf. on Comp. Aided Verf. (CAV)*, pages 72–83, 1997.

[14] A. Groce and W. Visser. Heuristics for model checking Java programs. *Int'l Jour. on Soft. Tools for Tech. Transfer*, 6(4):260–276, 2004.

[15] NASA. Java PathFinder, Version 4. In *http://javapathfinder.sourceforge.net*, 2007.

[16] D. Owen and T. Menzies. Lurch: a lightweight alternative to model checking. In *International Conference on Software Engineering and Knowledge Engineering*, pages 158–165, 2003.

[17] R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing random walk state space exploration. In *Int. Workshop on Formal Methods for Industrial Critical Systems*, pages 98–105, 2005.

[18] R. Pelánek and P. Šimeček. Estimating state space parameters. In *Proceedings of the 7th international Workshop on Parallel and Distributed Methods in Verification*, 2008.

[19] Robby, M. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. European Software Eng. Conf.*, 2003.

[20] N. Rungta and E. G. Mercer. Generating counter-examples through randomized guided search. In *SPIN Workshop on Model Checking of Software*, pages 39–57, 2007.

[21] H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. In *Electronic Notes Theor. Comput. Sci*, 2003.

[22] A. Taleghani. Using software model checking for software component certification. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 99–100, 2007.

[23] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[24] C. West. Protocol validation by random state exploration. In *Proc. of the 7rd Workshop on Protocol Specification, Testing and Verification*, 1986.