Random Testing and Model Checking: Building a Common Framework for Nondeterministic Exploration

Alex Groce
Laboratory for Reliable Software
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
Alex.Groce@jpl.nasa.gov

Rajeev Joshi
Laboratory for Reliable Software
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
Rajeev.Joshi@jpl.nasa.gov

The research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Funding was also provided by NASA ESAS 6G.

ABSTRACT

Two popular forms of dynamic analysis, random testing and explicit-state software model checking, are perhaps best viewed as search strategies for exploring the state spaces introduced by nondeterminism in program inputs. We present an approach that enables this nondeterminism to be expressed in the SPIN model checker's PROMELA language, and then lets users generate either model checkers or random testers from a single harness for a tested C program. Our approach makes it easy to compare model checking and random testing for models with precisely the same input ranges and probabilities and allows us to mix random testing with model checking's exhaustive exploration of nondeterminism. The PROMELA language, as intended in its design, serves as a convenient notation for expressing nondeterminism and mixing random choices with nondeterministic choices. We present and discuss a comparison of random testing and model checking. The results derive from using our framework to test a C program with an effectively infinite state space, a module in JPL's next Mars rover mission. More generally, we show how the ability of the SPIN model checker to call C code can be used to extend SPIN's features, and hope to inspire others to use the same methods to implement dynamic analyses that can make use of efficient state storage, matching, and backtracking.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publis or reproduce this article, or to allow others to do so, for Government purposes only.
WODA – Workshop on Dynamic Analysis, July 21, 2008
Copyright 2008 ACM 978-1-60558-054-8/08/07 ...$5.00.

Keywords

random testing, model checking, dynamic analysis, test frameworks

1. INTRODUCTION

Random testing [9] and model checking [2] can, at the highest level, be viewed as strategies for exploring state spaces created by nondeterminism in program input. Random testing explores such a state space by taking a large number of random walks through the graph. Model checking explores the state space more systematically, often using a depth-first search in the case of explicit state model checking of executable code. For example, in model-driven verification [11], a program is model checked by embedding calls to C code in a harness written in the SPIN model checker's PROMELA language [12]. SPIN generates a specialized model checker which performs a depth-first search (or variation thereof), backtracking the model checking harness and C program to earlier states as needed.

During efforts to apply random testing and model checking to JPL flight software we made two observations: (1) model checking is not *obviously* superior to random testing when searching effectively infinite state spaces, and (2) model checking is, in many cases, easier to set up than random testing. Both of these observations are at least somewhat contrary to conventional wisdom. Certainly, when exhaustive exploration is practical, model checking is preferable to random testing. The advantages of model checking are less obvious when exhaustive exploration of states is not possible: is a depth-first search (or another systematic search) of a small portion of a state space always superior to a series of random walks through that state space? For example, use of unsound abstractions in model checking may cause systematic exclusion of states that could at least potentially be explored using random testing, but is often necessary when dealing with rich properties of very large state spaces[1]. Hamlet has recently argued that there are indeed cases where "only random testing will do" [10]. Our second observation is also surprising: it is generally believed (including by us, as in our earlier work on random testing [6]) that random testing is "easier" than model checking, which is often a serious engineering challenge for real programs. We

[1]In our terminology, abstraction is unsound when we abstract by "lifting" concrete states produced by execution, but there is no guarantee that if s and s' abstract equivalently their successors will also do so. In this case, we may never explore some reachable abstract states.

discovered that, after we began using improved tools to prevent problems with backtracking [7], the primary difference between setting up model checking and setting up random testing for critical embedded code was that PROMELA is a better language for expressing nondeterministic choice than C (at least to our tastes), and the model checker provided considerable infrastructure, such as test playback, limitation of maximum test length, and support for iterative-deepening test case minimization that we were forced to implement ourselves when building a random tester. We also observed that using both model checking and random testing on a software system was often almost twice as much effort as applying only one method, even when the test design was essentially unchanged. Due diligence inclined us to apply both methods, but the redundant implementation effort was a drain on time better spent refining the test design (and running more tests).

Inspired by these observations, we developed a framework that allows us to use the same PROMELA harness to perform either random walks or searches with state matching and backtracking using the SPIN model checker. A key feature of our framework is that it does not require modifying SPIN itself in any way. Rather, we make use of a macro package for nondeterminism that uses SPIN's ability to embed C code in PROMELA to *modify the graph structure itself* (and thus the search used by the model checker). We believe that others interested in performing dynamic analysis of C programs using state matching and backtracking might benefit from this approach — and from using SPIN to produce program executions to analyze. SPIN is an extremely efficient model checker, the namesake of a long-running series of workshops on software model checking, and won the ACM System Software Award in 2002. It provides many powerful options for complete or partial exploration of C program state spaces, including various state hashing strategies, multi-core and distributed exploration, and state matching based on user-defined abstractions. In order to explore C program state spaces it is not necessary for SPIN users to become model checking experts or learn temporal logic. Because SPIN simply makes calls to C functions (or sets variables) to interface with the program being explored, there is no limit on program source size — SPIN can even model check binaries without source, so long as a public interface and memory locations of program state can be determined.

2. BUILDING THE FRAMEWORK

As indicated in the introduction, the SPIN model checker can be used to verify or test C programs. SPIN works by taking a harness (model) written in the PROMELA language and producing a specialized model checker, written in C, for that harness. Because SPIN generates C programs, including native C code in the transitions used to define the state space is easy. The only additional work required is to define the memory used by the C code so that the model checker can backtrack the program state space when exploring different input choices.

2.1 SPIN Nondeterminism

The SPIN model checker's PROMELA language includes a construct for expressing nondeterministic choice:

```
if
:: a      -> x = 1
:: a || b -> x = 2
:: a && b -> x = 3
:: else   -> x = 4
fi
```

Each sequence in the `if` consists of a guard and a statement to execute. There will be a successor to the `if` statement in the state space for every enabled guard from the previous state — the `else` is only enabled if no other guards are true. If more than one guard is true, the state with the `if` will have multiple successors, which will be explored during the state space search. A state in which neither `a` nor `b` is true will have one successor (`x = 4`), a state in which `b` alone is true will have one successor (`x = 2`), a state in which `a` alone is true will have two (`x = 1, x = 2`), and a state in which both `a` and `b` are true will have three successors (all but `x = 4`).

2.2 Expressing Wider Choice Ranges

The `if` notation is useful for expressing the limited nondeterminism found in the protocols and concurrent algorithms SPIN was originally designed to model check. In software model checking, however, it is often useful to succinctly express larger ranges of choice, i.e., the selection of a random file descriptor. Many SPIN models thus define a macro like this:

```
inline pick(var, range) {
  var = 0;
  do
  :: (var < range) -> var++
  :: break
  od;
}
```

The `do` construct works just like the `if` construct above, except that it loops until a `break` is chosen (there is an implicit guard of `true` for the `break` statement). The effect of the `pick` macro is to assign `var` to a value from 0 to `range`.

2.3 Using `pick` for Random Testing

When SPIN backtracks, it continues until it finds a nondeterministic choice[2]. If we change all nondeterminism to use `pick`, we can control backtracking, and force the model checker to behave as a random tester:

```
inline pick(var, range) {
  if
  :: !initialized ->
     nondet_pick(seed, SEED_RANGE);
     c_code{printf ("Test with seed %d\n",
                    now.seed);
            srandom(now.seed);};
     initialized = TRUE
  :: else -> skip
  fi;
  var = c_expr{random()} % (range + 1);
}
```

[2]Or a choice in process scheduling; however, in this paper we focus on a single process, as most testing of C programs with SPIN does not involve concurrency.

where nondet_pick is just the earlier pick macro, c_code allows us to include arbitrary C code in our model, and now.seed is the notation for accessing a PROMELA variable in C code. We can see the different behaviors of the two macros easily with a trivial PROMELA model:

```
n = 0;
do
:: n < 4 ->  pick(x,100);
             c_code{printf ("[%d: x = %d] ",
                            now.n, now.x);};
             n++
:: else -> break
od;
c_code{printf ("\n");}
```

The loop picks 4 integers. Exploration using the first pick macro produces this output:

```
[0: x = 100] [1: x = 100] [2: x = 100] [3: x = 100]
[3: x = 99]
[3: x = 98]
...
[3: x = 0]
[2: x = 99] [2: x = 98] [2: x = 97] [2: x = 96] ... [1: x = 99]
```

Using our second pick macro, we get something like this:

```
Test with seed 100
[0: x = 40] [1: x = 1] [2: x = 79] [3: x = 84]
Test with seed 99
[0: x = 72] [1: x = 24] [2: x = 49] [3: x = 13]
Test with seed 98
[0: x = 21] [1: x = 67] [2: x = 75] [3: x = 43]
...
```

In the first case, the model checker chooses x = 100 four times, then backtracks to explore all possibilities for the last choice (where n = 3) before trying alternative values for the next to last choice (where n = 2). Because the final choice of x overwrites that choice, SPIN backtracks and so no output is produced beyond the point where n = 2. When all choices at n = 2 have been explored, the model checker will similarly explore all choices at n = 1. The output for the second pick macro is simply a sequence of random tests, with seeds for the random number generator decreasing from 100 to 0.

The second macro works by moving all the nondeterminism to an initial choice of a random seed. When the model checker backtracks, it is forced to unwind the entire random test and pick a new seed. We've changed the state space itself to represent a set (fixed by the seed range) of random walks through the original nondeterministic state space.

Without modifying SPIN, we have changed the model checker generator into a random tester generator. We can now use the same models for both model checking and random testing, enabling us to compare search techniques and making it convenient to apply both strategies (in the spirit of search diversification, somewhat like Dwyer et al.'s approach [3]) to one model without writing two test harnesses.

2.3.1 *Using* pick *to Permute Choices*

A useful application of the same trick is to permute choices when using full nondeterminism in model checking. As shown in the output above, the usual do loop explores choices in sequential order. When the range is large and complete state-space exploration is impossible, this unfortunately biases our exploration to one portion of a range, especially if several choices from the same range are made in a loop. It would be better, even when performing model checking, to explore each choice once but to permute the choices themselves. As shown by Dwyer et al., this search order can be a controlling factor in whether the model checker finds an error in a large state space [4]. Producing permutations of nondeterministic choices is in fact quite easy, using the same method we used to introduce random testing. The core of the approach is quite simple: use a fixed seed to randomly permute a choice array at each call to pick, then return a value chosen from that array by nondet_pick:

```
c_code {shuffle(choices);}
nondet_pick(i, range);
var = c_expr{choices[now.i]};
```

```
inline pick(var, range) {
  if
  :: !initialized ->
     nondet_pick(seed, SEED_RANGE);
     c_code{printf ("Test with seed %d\n",
                    now.seed);
            srandom(now.seed);};
     initialized = TRUE
  :: else -> skip
  fi;
  c_code{srandom(now.last_seed);};
  var = c_expr{random()} % (range + 1);
  c_code{now.last_seed = random();};
}
```

Figure 1: The pick macro, fixed to allow mixing random choice with full nondeterminism

2.4 Mixing Random Choice with Full Nondeterminism

Given that we are performing our random testing with a model checker, it would be nice to mix random testing with full nondeterminism. For example, we might want to generate operations and inputs for testing a file system randomly (because the full range of choices is far too large to exhaustively explore), but exhaustively explore the placement of system resets in each such test. It would be useful if our framework made it easy to experiment with this kind of hybrid search strategy. Unfortunately, as written our pick macro won't quite work. We can indeed mix if nondeterminism (or nondet_pick calls) with pick calls, but when the model checker backtracks the program to a nondeterministic choice, it will not be able to backtrack the state of the system's random number generator, and so different choices will be made than in the first "version" of the test. Fortunately, we can easily remedy this by using a PROMELA variable to seed the random number generator at each call to pick (Figure 1).

The model checker will restore the old value of last_seed whenever the state is backtracked, allowing us to mix nondeterminism and random choice as we wish.

Figure 2 shows the heart of a simple model mixing randomizeable nondeterminism using pick with full nondeterminism. If we use nondet_pick in place of pick, and set MAX to 1,000, MIN to 800, and T to 5, the model checker explores 10,628,619 choices for t and k2 before finding an assertion violation; this is improved considerably to "only"

```
n = 0;
pick(k1, MAX);
if
:: t = 0
:: t = 1
fi;
do
:: (n < T) -> pick(k2, MAX);
              c_code {printf ("A%d:  %d(%d/%d)\n", now.n,
                                    now.t, now.k2, now.k1);};
              assert (t || ((k1-k2) < MIN));
              k1 = k2;
              n++;
:: else -> break
od;
n = 0;
if
:: t = 0
:: t = 1
fi;
do
:: (n < T) -> pick(k2, MAX);
              c_code {printf ("B%d: %d(%d/%d))\n", now.n,
                                    now.t, now.k2, now.k1);};
              assert (!t || ((k1-k2) < MIN));
              k1 = k2;
              n++;
:: else -> break
od;
c_code {printf ("DONE\n");};
```

Figure 2: Core of `example.pml`

4,011,045 choices if we randomly permute the choices. Random testing, in contrast, explores only 13 choices! Using mixed nondeterminism, the tester performs only 1 "test," which violates the assertion. It is instructive to look at the output of the tester (Figure 3). The tester first explores a sequence of choices in both loops. It then backtracks to the B loop, and tries the *same* choices with `t` set to 1. If this hadn't found an assertion violation, the model checker would have backtracked again and tried the same choices in the A loop with `t` set to 1. If we change the harness to use pure random testing, the results change slightly but random testing still outperforms model checking by many orders of magnitude.

Why does random testing do so much better than model checking in this case? The key is the wide range of choices for `k2` – the model checker is performing a depth-first search, which means that it has to try all 1,000 choices for `k2` in the last step of the run before it can reconsider the previous choices. Unfortunately, the error cannot be exposed in the second loop until the model checker backtracks all the way to the decision on `t`. This difficulty is not addressed by the usual reduction technique of introducing an (unsound) abstraction, if we abstract by matching states rather than by limiting inputs: the model checker still has to generate all the possible choices, even if only to discard them as reaching an already visited abstract state. Permuting the order of the nondeterministic choices is also ineffective — the choices must all be explored, whatever the order. Switching to breadth-first search often only makes the problem worse, due to the memory requirement for the state queue, though in some cases a heuristic search such as A* may be able to address the issue of large choice ranges [8]. The example is contrived, but the problem is real. In general, users try very hard to avoid using large choice ranges in explicit-state model checking.

This example serves as a trivial analogue for the possibility

```
Test with seed 100001
A0:  0(870/309)
A1:  0(131/870)
A2:  0(940/131)
A3:  0(706/940)
A4:  0(790/706)
B0:  0(606/790)
B1:  0(984/606)
B2:  0(122/984)
B3:  0(998/122)
B4:  0(67/998)
DONE
B0:  1(606/790)
B1:  1(984/606)
B2:  1(122/984)
pan: assertion violated ( !(t)||((k1-k2)<800)) (at depth 148)
```

Figure 3: Output for random testing with mixed nondeterminism

of checking a file system with random testing for generating operations and parameters mixed with full nondeterminism for choosing placement of system resets.

2.5 Boolean Choice and Probabilities

We also provide a `pick_bool` macro, allowing users to assign a boolean choice without specifying a range of 1. This macro takes three arguments: `pick_bool(var, num, den)`. The parameters `num` and `den` indicate the probability that the variable should be assigned true. We are exploring convenient notations for allowing biased choice to be introduced into the original `pick` macro, but find that concise but flexible expression of probability distributions is not easy.

3. APPLICATIONS: FLIGHT SOFTWARE DATA STORAGE MODULES

We applied our framework to a large-scale effort to model check and test flight software systems used for data storage (on flash memory and in a RAM file system) for NASA/JPL's upcoming Mars Science Laboratory mission [1]. In earlier work, we described our application of random testing with feedback to a preliminary version of these modules [6], and our use of automated source code instrumentation to model check the lowest-level (and most critical) flash storage module [7]. In this paper we report on our application of a unified framework for random testing and model checking to the full POSIX flash file system. The model is similar to the NVDS (Non-Volatile Data Storage) model described in earlier work, except that the range of operations and inputs is extended to the POSIX operations used in the flight file system, including choice of file descriptor and path names. We make use of the same approach to automatically instrument the code so we can detect modifies clause (i.e., memory safety) violations. Figure 4 shows a simplified version of the part of the PROMELA harness that tests `unlink`. The pseudo-PROMELA above is not in any way specific to MSL or space usage: this is what any tester for POSIX file systems would look like, at a high level. As shown, we determine the correctness of NVFS (Non-Volatile File System) behavior by comparing results with our RAMFS (RAM File System) implementation for MSL, which has been more thoroughly tested via comparison with Linux file systems (we are not injecting RAMFS with faults — it operates without experiencing the resets and hardware failures our harness presents to NVFS) and with another, independently developed, RAM

```
:: choice == UNLINK -> /* unlink */
   pick(pathindex, NUM_PATHS);  /* Choose a path */
   c_code { enter_nvfs(); /* Allow memory access to NVFS region */
            now.res = nvfs_unlink (path[now.pathindex]);
            now.nvfs_errno = errno;
            leave(); /* Disallow memory access */ };
   check_reset(); /* Check for system reset and reinitialize/mount NVFS if needed */
   if
   :: (res < 0) && (nvfs_errno == ENOSPC) -> /* If there was an out-of-space error */
      check_space();
   :: ((!did_reset) || (res != -1)) && !((res < 0) && (nvfs_errno == ENOSPC)) ->
      c_code{ enter_ramfs(); /* Allow memory access to RAMFS region */
              now.ramfs_res = ramfs_unlink (path[now.pathindex]);
              now.ramfs_errno = errno;
              leave(); /* Disallow memory access */ }
   :: else -> skip
   fi;
   ...
   assert (res == ramfs_res);
   assert (nvfs_errno == ramfs_errno);
```

Figure 4: Simplified PROMELA code for file system testing

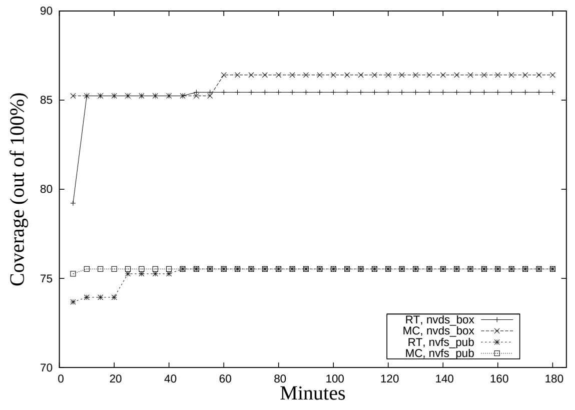

Figure 5: Source code (statement) coverage

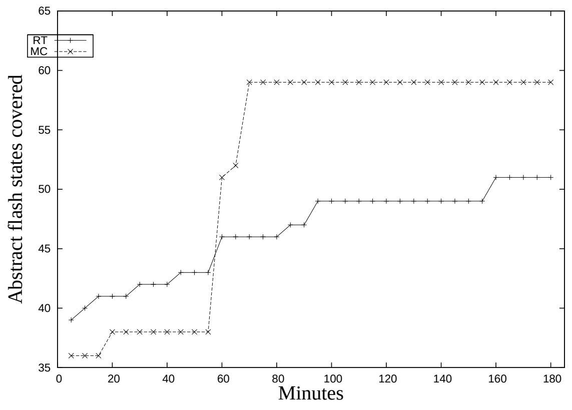

Figure 6: Abstract flash state coverage

file system, a standard use of differential testing [13]. The testing and model checking has revealed a number of important bugs, in both RAMFS and NVFS. The full PROMELA harness is about 2,700 lines, and is considerably more readable and flexible than the test driver used in earlier random testing (about 5K lines of C).

NVFS itself is a C program of approximately 8K lines. Its state space, even when unsound abstractions are applied and flash space is reduced to 8 blocks of 4 pages with 112 bytes of data, is effectively "infinite" — bitstate model checking on a 32GB system suggests a lower bound of at least 5×10^{10} states.

4. EXPERIMENTS: RANDOM TESTING VS. MODEL CHECKING

Our framework made it possible to experiment with exploring the same (effectively infinite) state space, with the same model and probabilities for nondeterministic choices via model checking and random testing. We compiled two model checkers from a SPIN harness for the MSL NVFS modules, with only one difference: in the first version, `pick` was defined to produce complete nondeterminism with random ordering of transitions; in the second version, the macro was defined to induce random testing with test runs of length 1,000. For model checking, we used an arbitrary depth-first search limit of 2,000 states[3].

We then executed both versions, in each case with a fixed time limit: that is, for each experiment we executed n minutes of model checking and n minutes of random testing. We began with a time limit of only 5 minutes, and increased the limit in steps of 5 minutes to 3 hours (180 minutes). The number of states stored by the model checker increased from just 6,000,000 states with a 5 minute limit to 1.84×10^8 with 180 minutes. For random testing, 5 minutes proved sufficient to perform 11,018 test runs. Increasing the time to 180 minutes allowed us to perform 375,512 runs. Of course, states stored and test runs are not directly comparable; these figures serve only to indicate how both methods scaled with additional run-time. As the time and number of system states stored or random tests performed increased, we expected that coverage of the module's source code and of abstractions of the system state would also increase.

4.1 Source Code Statement Coverage

Figure 5 shows how statement coverage of the most interesting source files (`nvds_box.c` and `nvfs_pub.c`) increased

[3]The state limit and test run length are not directly comparable: the length in the first case is measured in number of random choices, the other in internal model checker states.

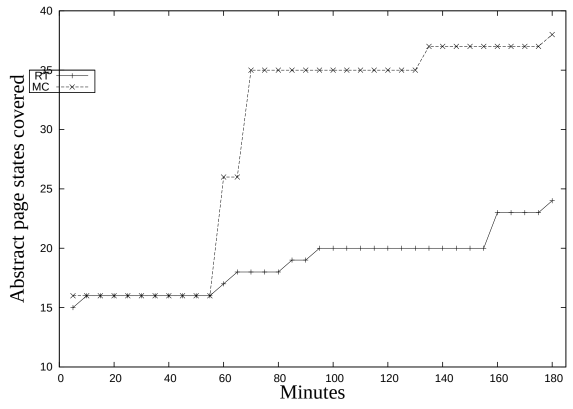

Figure 7: Abstract page state coverage

over time. Model checking began with significantly higher coverage and covered at least as many statements for almost every time limit, in both cases. For the NVFS public interface (`nvfs_pub.c`), random testing eventually matched the coverage level of model checking. For this portion of the code, we believe coverage to be complete: that is, we do not believe that the choices allowed by the harness make it possible to cover more of the source code. The second file, `nvds_box.c`, is more interesting, as we do not know if the coverage was complete at 86.41% for model checking. This file contains much of the low-level behavior managing how "boxes" (files and directories) are stored on flash. For this critical file, random testing not only initially covered less code than model checking, but the gap remained significant even at 3 hours (though random testing did improve on model checking at the 50 and 55 minute limits).

We speculate that the coverage of the public interface is governed almost purely by the choice of parameters: while the state of the file system does matter (for instance, covering code returning an error code for a file that does not exist), it only matters in a "shallow" way. Obtaining complete coverage requires knowledge only of the tree structure of the file system and the valid ranges of parameters, and random operations will usually produce states in which it is easy to "guess" both a pathname that exists and a pathname that does not exist. For `nvds_box.c`, on the other hand, coverage depends on the state of the flash device — which pages are in use and dirty and which blocks have the highest wear counts. An alternative way of looking at this distinction is that we expect that it would be possible (or even easy) to cover `nvfs_pub.c` by applying random operations to almost any random file system state. For a fixed file system state it would be impossible to generate inputs to cover `nvds_box.c`, as some conditions are mutually exclusive (e.g., the flash device cannot be both full and nearly unused), and even for code paths reachable from that state inputs might be unique, rather than members of a large equivalence class (consider the difficulty of guessing "write between 50 and 60 bytes to file *f*" vs. "attempt to open a file that does not exist"). Because model checking backtracks to explore all possibilities and stops exploration when it reaches an already visited state, it *should* have an advantage in obtaining coverage.

For other files, `nvds_tree.c`, `nvds_nvm.c` and `nvds_ftl.c`, coverage essentially did not vary with increased time or change in exploration method.

4.2 Abstract Flash State Coverage

An independent measure of a search of this state space is the number of *abstract states* covered. One abstraction we use is based on how flash memory is being used. For each block of the flash device, we consider only the module's notion of the block state (is the block in use, free, or dead?) and the number of live and dirty pages on the block. Figure 6 shows how model checking and random testing improved their coverage of this abstraction. Here the story is more complex than with source code coverage: even though the model checker bases its exploration on this abstraction (never backtracking when it has found a new abstract state), random testing covers *more* abstract states until around the 50-minute mark. After this point, however, model checking consistently covers more states, including very difficult-to-produce scenarios resulting from precisely-timed system resets. We do not know if the coverage is complete, but have not observed improved coverage even for multi-day runs on a 32GB machine.

The shapes of the graphs for a similar but finer-grained abstraction based on the states of *all pages* on the flash (using 32 rather than 8 abstract variables) are similar (Figure 7). In this case, however, model checking begins with better coverage, and random testing briefly matches but never improves on model checking.

5. RELATED WORK

Visser et al. implemented a true random testing mode [16] in Java PathFinder [15], in order to compare exhaustive model checking (with or without symbolic execution), random testing, and model checking with lossy abstractions, for basic block and predicate coverage of Java data structures. Their implementation does not allow mixing of random testing mode and full nondeterminism with backtracking (though this would be fairly easy to implement in JPF, using a choice generator). The key differences with our work are that JPF has the advantages and disadvantages of Java rather than PROMELA (with as much C as is needed or desired) as a language for a test harness, and that JPF is a considerably less efficient model checker than SPIN, even if large fragments of native Java are used.

Dwyer et al. note the importance of transition ordering in model checking, and establish the difficulty of evaluating different search techniques [4]. Concern about high sensitivity to error location and search order influenced our decision to report coverage results in our experimental results rather than compare the error detection effectiveness of random testing and model checking.

6. CONCLUSIONS AND FUTURE WORK

Given our uncertainty about how best to test programs with very large state spaces, we would like to apply both systematic techniques (such as model checking) and random testing. Unfortunately, the effort required to produce a model checking harness and a random tester is often close to twice that required to produce just one of these. In this paper, we show how to use a model checking harness to perform true random testing. Our framework also makes it possible to mix model checking and random testing, giving us the tools to explore novel and potentially useful search strategies that are difficult or impossible to use with pure model checking or pure random testing.

6.1 Is Model Checking Better than Random Testing?

For our example, assuming we have more than an hour to test a program, model checking appears to cover the state space more effectively than random testing. It would be premature to draw any general conclusions from one non-trivial real world application such as our file system testing. The results of Visser et al. show that random testing can (in some circumstances) be competitive with exhaustive exploration [16]. In both cases, it might be better to compare with random testing using feedback [6, 14] to limit the choices available based on past history, mitigating random testing's tendency to execute many redundant and invalid operations. An example of using feedback in this case would be to limit pathname choices in a state to the set of pathnames provided as arguments to successful `mkdir` or `creat` operations (a set that would initially contain only the root path `/`), with the possibility of adding one additional random pathname component. E.g., if the history set was `{/, /a, /b}`, path choices would include the members of the set plus `/c`, `/d`, `/a/a`, `/a/b`, and so forth, but *not* `/a/b/c` or `/c/a`. We restrict the choices based on the observation that if the file system is correct, no POSIX operation can ever succeed on a path that is not of this form. We do not remove paths from the history when they are deleted from the file system, as the "resurrection" of dead files is a common fault. Feedback is also potentially useful for model checking, though the model checker will likely benefit less because of state-matching. It is also the case that our model was originally designed with model checking in mind: we therefore greatly limited the choices for pathnames, file descriptors, placement of simulated hardware faults, and number of bytes in arguments to `read` and `write`. Random testing is (as shown in the toy example from Section 2.4) often more effective than model checking when a depth-first search must consider each of 1,000 possibilities for the last choice made in a path before it is able to re-examine earlier choices. Our model is easily parametrized, so we hope to perform a much more in-depth study of how the range of parameters and use of feedback affects the effectiveness of both model checking and random testing. Preliminary results for experiments in which we increase the range of bytes written and read from 5 to over 300 show random testing as slightly superior in statement coverage for some files, but remaining generally less effective as test time increases.

6.2 Inspiring New Approaches to Model Checking and Dynamic Analysis

More generally, we hope that our method, which changes the shape of the state space without modifying the model checker — essentially implementing new search strategies by using SPIN's ability to call native C code — will inspire other researchers to "hack" the model checker and introduce heuristic search and other capabilities useful for testing but missing from SPIN's standard release.

Finally, we hope that this approach to random testing will inspire other dynamic analysis and testing researchers to consider the use of SPIN for analyzing C programs. At least for embedded systems with clearly defined state memory, the process is relatively painless and enables a wide variety of partial and exhaustive selections of executions to explore. Because SPIN works by actually executing C code, using dynamic analysis tools such as Daikon [5] with the model checker generating the analyzed executions is often trivial.

7. REFERENCES

[1] http://mars.jpl.nasa.gov/msl/.

[2] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[3] Matthew B. Dwyer, Sebastian G. Elbaum, Suzette Person, and Ragul Purandare. Parallel randomized state-space search. In *International Conference on Software Engineering*, pages 3–12, 2007.

[4] Matthew B. Dwyer, Suzette Person, and Sebastian Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Foundations of Software Engineering*, pages 92–104, 2006.

[5] Michael Ernst, Jake Cockrell, William Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[6] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[7] Alex Groce and Rajeev Joshi. Extending model checking with dynamic analysis. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 142–156, 2008.

[8] Alex Groce and Willem Visser. Heuristics for model checking Java programs. *Software Tools for Technology Transfer*, 6(4):260–276, 2004.

[9] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[10] Richard Hamlet. When only random testing will do. In *International Workshop on Random Testing*, pages 1–9, 2006.

[11] Gerard Holzmann and Rajeev Joshi. Model-driven software verification. In *SPIN Workshop on Model Checking of Software*, pages 76–91, 2004.

[12] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[13] William McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.

[14] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

[15] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[16] Willem Visser, Corina Păsăreanu, and Radek Pelanek. Test input generation for Java containers using state matching. In *International Symposium on Software Testing and Analysis*, pages 37–48, 2006.