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

LARS
NASA/JPL Laboratory for Reliable Software

Jet Propulsion Laboratory,
California Institute of Technology

Alex Groce and Rajeev Joshi

Background & Motivation

- LaRS (Laboratory for Reliable Software) at JPL has been building, verifying, and **testing** flash file systems for space mission use

- This work grows out of that experience

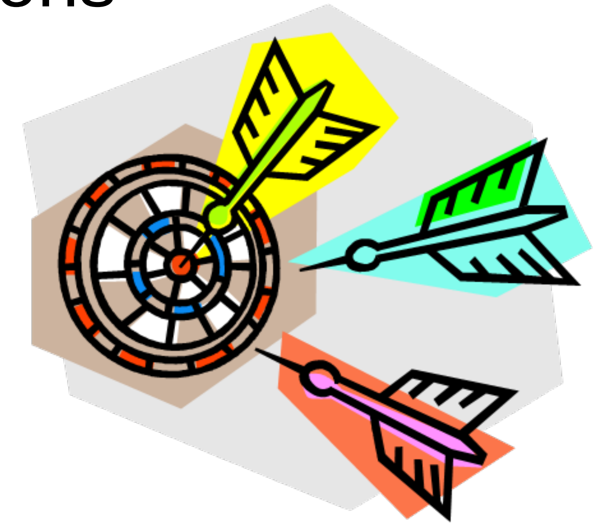
Background & Motivation

- MSAP
  - Two flash file systems, one RAM file system, one critical parameter storage module
  - Approach: **random testing** [ICSE'07,ASE'08]

- MSL (Mars Science Laboratory)
  - One flash file system, one RAM file system, one low-level flash interface (critical parameter storage)
  - Approach: **model checking/random testing**

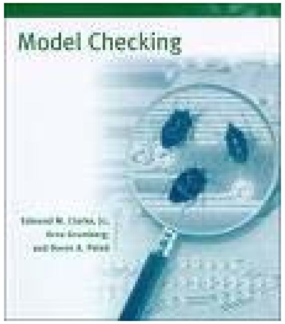
Random Testing

- I think we all know what random testing is:
  - Operations and parameters generated at random to test a program
  - Possibly with some bias or feedback to help with the problem of irrelevant/redundant operations



Model Checking and Dynamic Analysis

- (Software) *model checking*
  - (In principle exhaustive) exploration of a program’s state space


Model Checking

- *Dynamic analysis* *(what we’re here for today)*
  - Analysis of a running program
  - Usually instrumentation or execution in virtual environment – e.g. Valgrind, Daikon
  - *Testing is a dynamic analysis*: program is executed in order to learn about its behaviors
  - We’re looking at the kind of model checking that is essentially a dynamic analysis

Many Software Model Checkers

BLAST

CRunner SPIN

CMC

CBMC JPF2

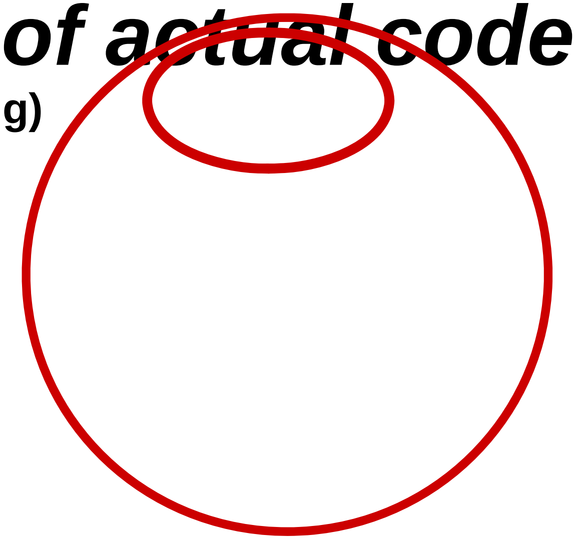
SLAM MAGIC

Bogor VeriSoft

Two Approaches

Execution of actual code

(dynamic: like testing)



BLAST

CRunner

SPIN

CMC

CBMC

JPF2

Our focus in this talk

SLAM

MAGIC

Bogor

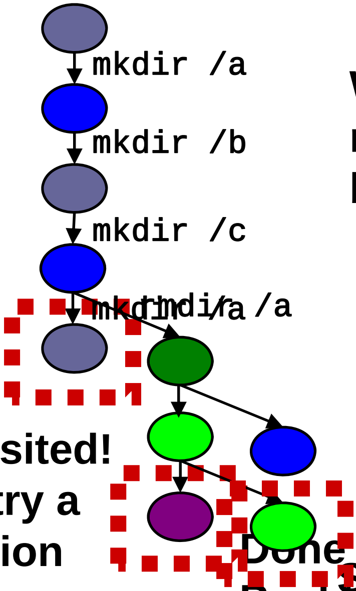
VeriSoft

Analysis of derived transition system

(“static”)

Model Checking as State-Based Testing

- Model-checking by *executing the program*
  - Backtracking search for all *states*

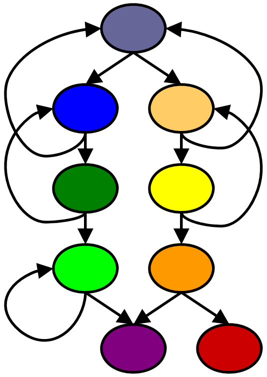


Will explore, as a side-effect, many executions (like random testing) but the goal is to explore states

State already visited! Backtrack and try a different operation

Done with test! Backtrack and try a different operation

State already visited! Backtrack and try a different operation



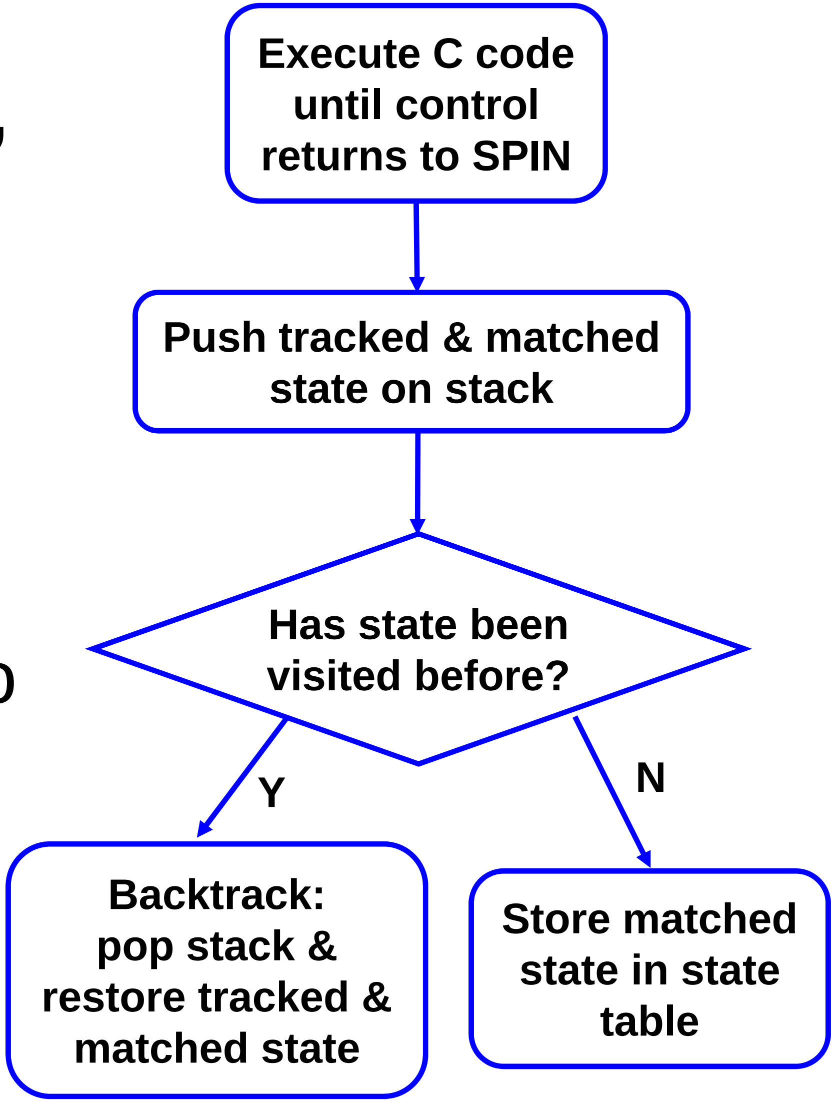
CFG

SPIN and Model-Driven Verification

- SPIN compiles a PROMELA model into a C program: it's a *model checker generator*
  - Embed C code in transitions by *executing* the compiled C code
  - Take advantage of all SPIN features – hashing, multicore exploration, etc.
- Requires the ability to restore a running program to an earlier execution state
  - Difficult engineering problem, handled by CIL-based automatic code instrumentation [VMCAI'08]

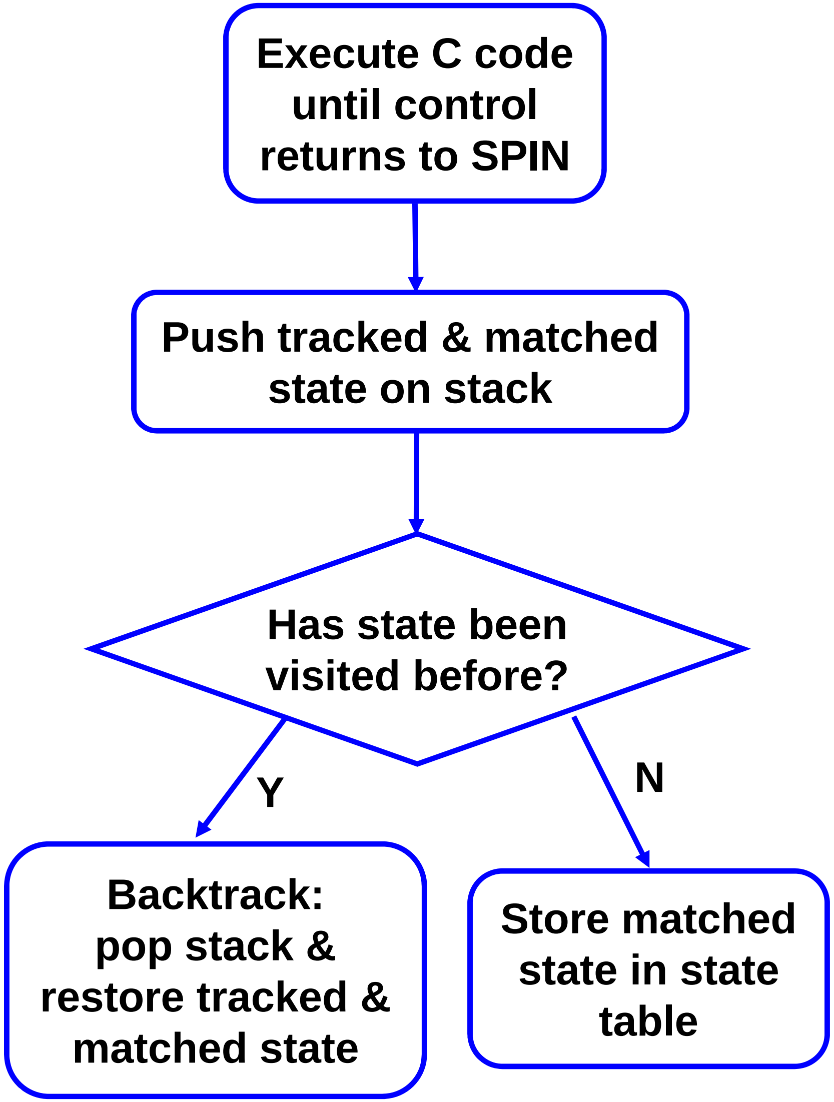
SPIN and Model-Driven Verification

- When SPIN backtracks, it uses information on how to restore the state of the C program:
  - *Tracked* memory is restored on backtrack
  - *Matched* memory is also used to determine if a state has been visited before


Execute C code until control returns to SPIN
Push tracked & matched state on stack
Has state been visited before?
Y
N
Backtrack: pop stack & restore tracked & matched state
Store matched state in state table

SPIN and Model-Driven Verification

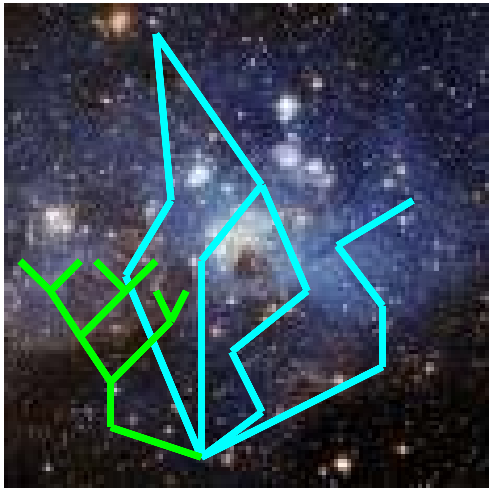
- (Unsound) abstraction by matching on an abstraction of the tracked concrete state
  - E.g. track the pointers/contents of a linked list
  - Match on a sorted array copy only (if order doesn't matter for property in question)


Execute C code until control returns to SPIN
Push tracked & matched state on stack
Has state been visited before?
Y
N
Backtrack: pop stack & restore tracked & matched state
Store matched state in state table

A Common Goal

- Program state spaces are typically too large to explore fully even after (unsound) abstraction

- Random testing and model checking are both methods for *nondeterministically exploring a program's state space*
  - A series of random walks
  - vs. systematic exploration with backtracking



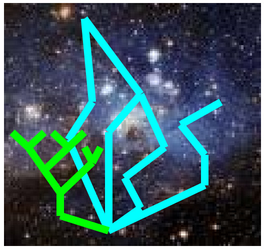
Which is Better?

- Conventional wisdom (exaggerated):
  - Random testing is probably less effective than model checking
  - BUT model checking is *much* more difficult to apply than random testing, scales poorly, crashes a lot, makes your ears bleed, and may cause temporary paralysis

Test engineer using a model checker on a C program?

How True is the Conventional Wisdom?

- Realistically, the state spaces for real programs are huge
  - Model checking will almost certainly use unsound abstractions, and *still* be only partial exploration
  - Systematically missing some states that could expose errors
  - Are we *sure* this is better than smart random testing for fault detection / coverage?



How True is the Conventional Wisdom?

- On the other hand, explicit-state model checking is not that difficult to apply
  - PROMELA is a nice language for expressing nondeterministic choice & test structure
  - Provides test-case playback, minimization, and other things often build by hand for testing
  - Scales quite well if memory usage is (a) limited (no 5GB memory footprint) and (b) well-defined
    - Often true for embedded systems

Using SPIN for True Random Testing

- Want to apply **both** methods
  - For research purposes (comparison)
  - Due diligence in testing! This stuff is going to Mars…
- But why write two testers? – one for random testing, one for model checking
  - Basic harness looks the same, property checks look the same, etc.
  - **Annoying redundant work**, better to spend time improving the harness or running more tests

UNITED STATES DEPARTMENT OF REDUNDANCY DEPARTMENT
CONTRARIA CONTRARIIS CURANTUR

A Quick Primer: Using SPIN for Random Testing, in Five Slides OR Almost All the PROMELA You Ever Need to Know

Simple PROMELA Code

```
int x;

int y;

active proctype main () {
 1
  if
 2 :: x = 1
 3 :: x = 2
   fi;
 5
   assert (x == y);
 7
}
```

Start simple

This model has 7 states

What are they?

State = (PC, x, y)

SPIN’s *nondeterministic choice* construct

Picks any one of the choices that is *enabled*

Not mutually exclusive!

How do we *guard* a choice?

```
if
:: (x < 10) -> y = 1
:: (x < 5)  -> y = 3
:: (x > 1)  -> y = 4
fi;
```

Simple PROMELA Code

```
int x;
int y;
active proctype main () {
  1 if
  2   :: x = 1
  3   :: x = 2
    fi;
  5 if
      :: y = 1
  7   :: y = 2
  9 fi;
 13 if
 14   :: x > y -> x = y
      :: y > x -> y = x
 15   :: else -> skip
 17 fi;
    assert (x == y);
}
```

This model has 17 states

What are they?

State = (PC, x, y)

Er…

Don't worry about state-counting too much – SPIN has various automatic reductions and atomicity choices that can make that difficult

Simple PROMELA Code

```
int x;


active proctype main () {
   x = 0;
   do
   :: (x < 10) -> x++
   :: break
   od
   /* Here, x is anything between
      0 and 9 inclusive */
```

Only a couple more PROMELA constructs to learn for building test harnesses: the do loop

Like `if`, except it introduces a loop to the top – `break` choice can exit the loop

This nondeterministically assigns x a value in the range 0…9

Simple PROMELA Code

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

`inline` gives us a macro facility

As you can imagine, this is a useful macro for building a test harness!

Less Simple PROMELA Code

```
:: choice == UNLINK -> /* unlink */
     pick(pathindex, NUM_PATHS); /* Choose a path */
     c_code {
             now.res = nvfs_unlink (path[now.pathindex]);
            };
     nvfs_errno = c_expr{errno};
     check_reset(); /* Check for system reset and reinit if needed */
    if
    :: (res < 0) && (nvfs_errno == ENOSPC) -> /* If out-of-space error */
       check_space();
    :: ((!did_reset) || (res != -1)) && !((res < 0) && (nvfs_errno == ENOSPC)) ->
       c_code{
              now.ramfs_res = ramfs_unlink (path[now.pathindex]);
             };
       ramfs_errno = c_expr{errno};
    :: else -> skip
    fi;
    ...
    assert (res == ramfs_res);
    assert (nvfs_errno == ramfs_errno);
```

Finally, we want to be able to call the C program we are testing

Testing via Model Checking

- Basic idea:
  - We’ll write a *test harness* in PROMELA
  - Use SPIN to backtrack and explore inputs
  - Use abstraction to limit the number of states we consider
  - We can even “trick” SPIN into doing pure random testing!

The pick Macro, Revisited

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

What if we change pick?

The pick Macro, Revisited

```
inline pick (var, MAX) {

  if
  :: ! initialized ->
     nondet_pick(seed, SEED_RANGE);
     c_code{
             printf (“Test with seed %d\n”,
                     now.seed);
             srandom(now.seed);
           };
     initialized = 1
  :: else -> skip
  fi;
  var = c_expr{random()} % MAX;

}
```

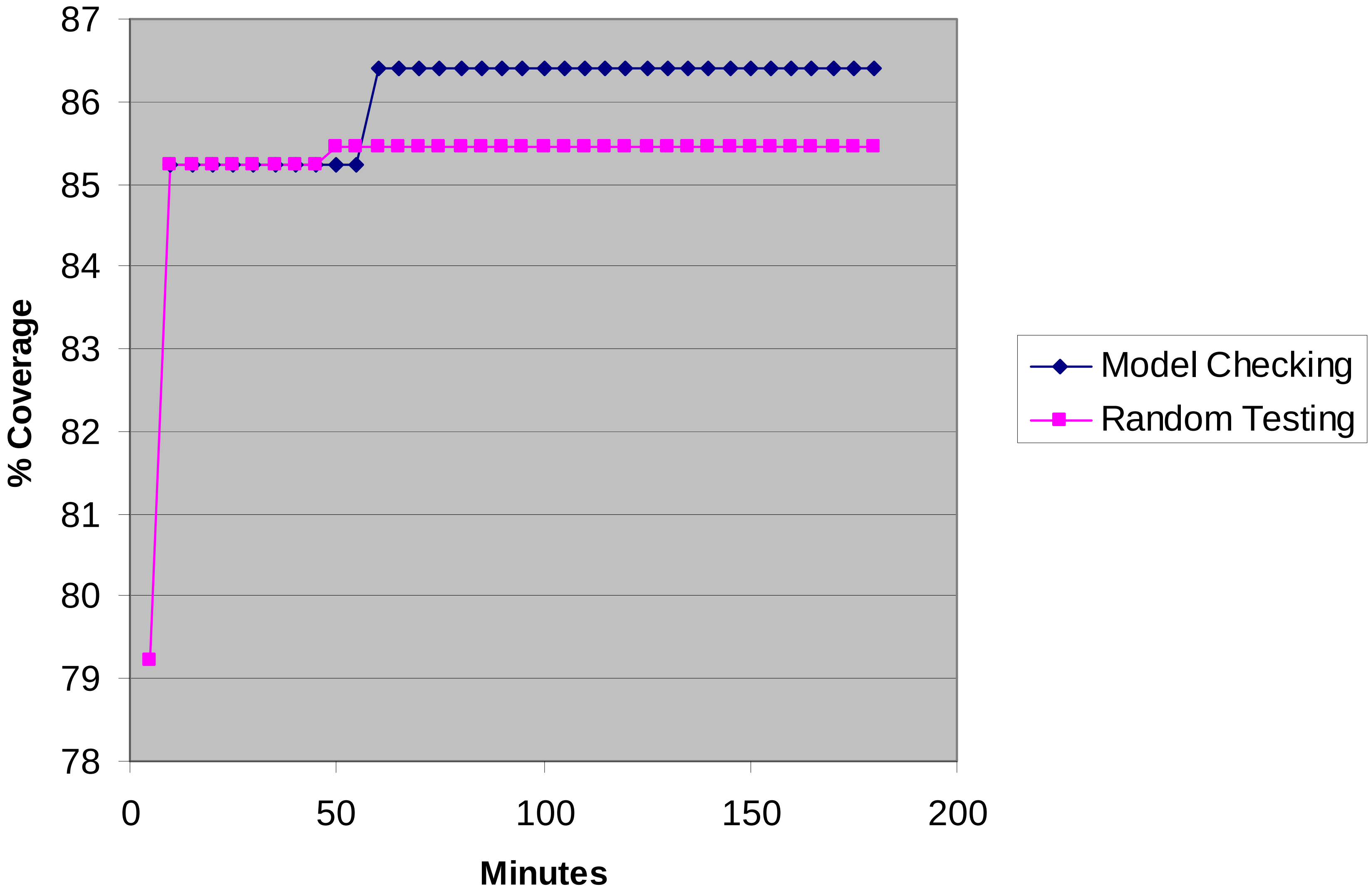
To this?

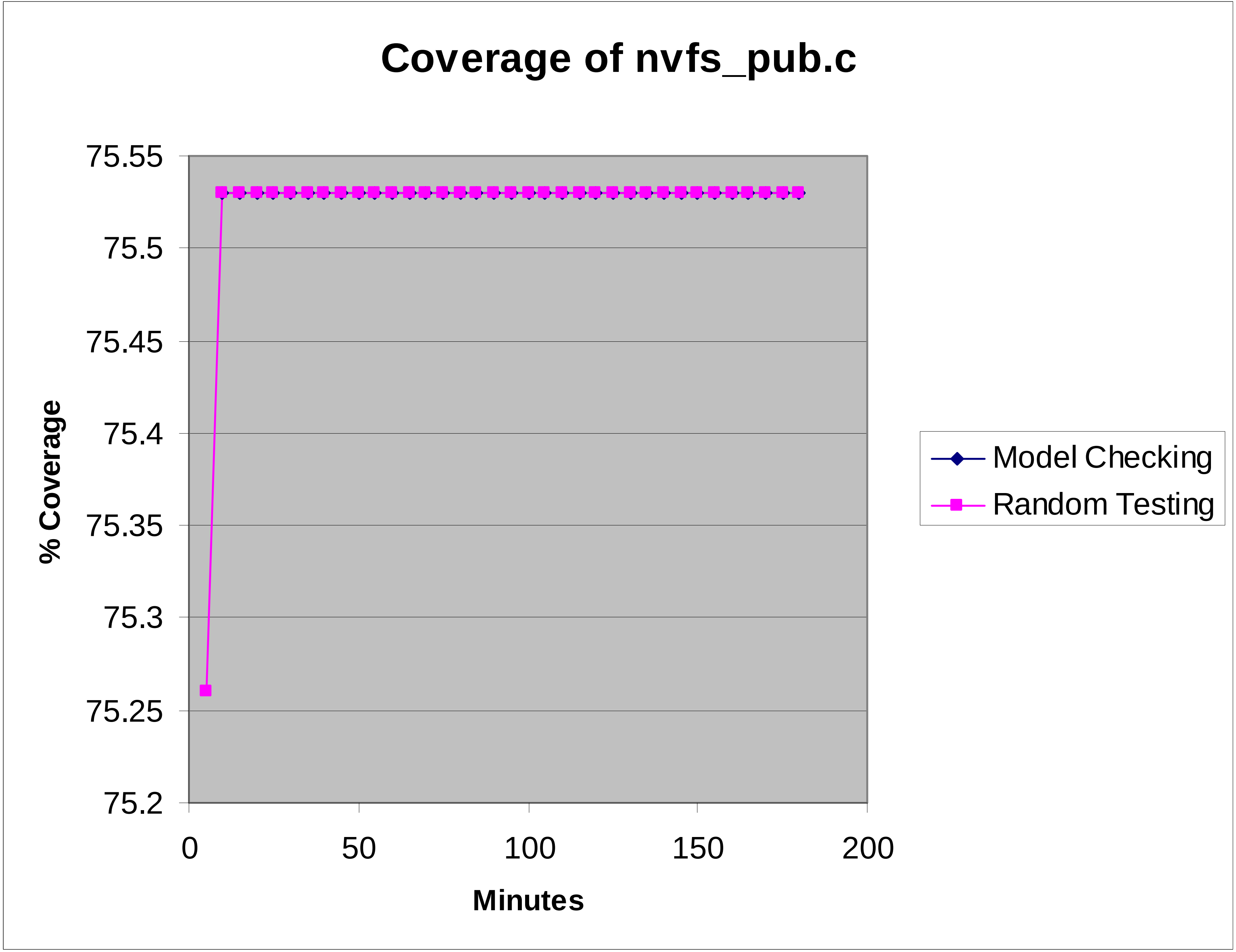
Some Results

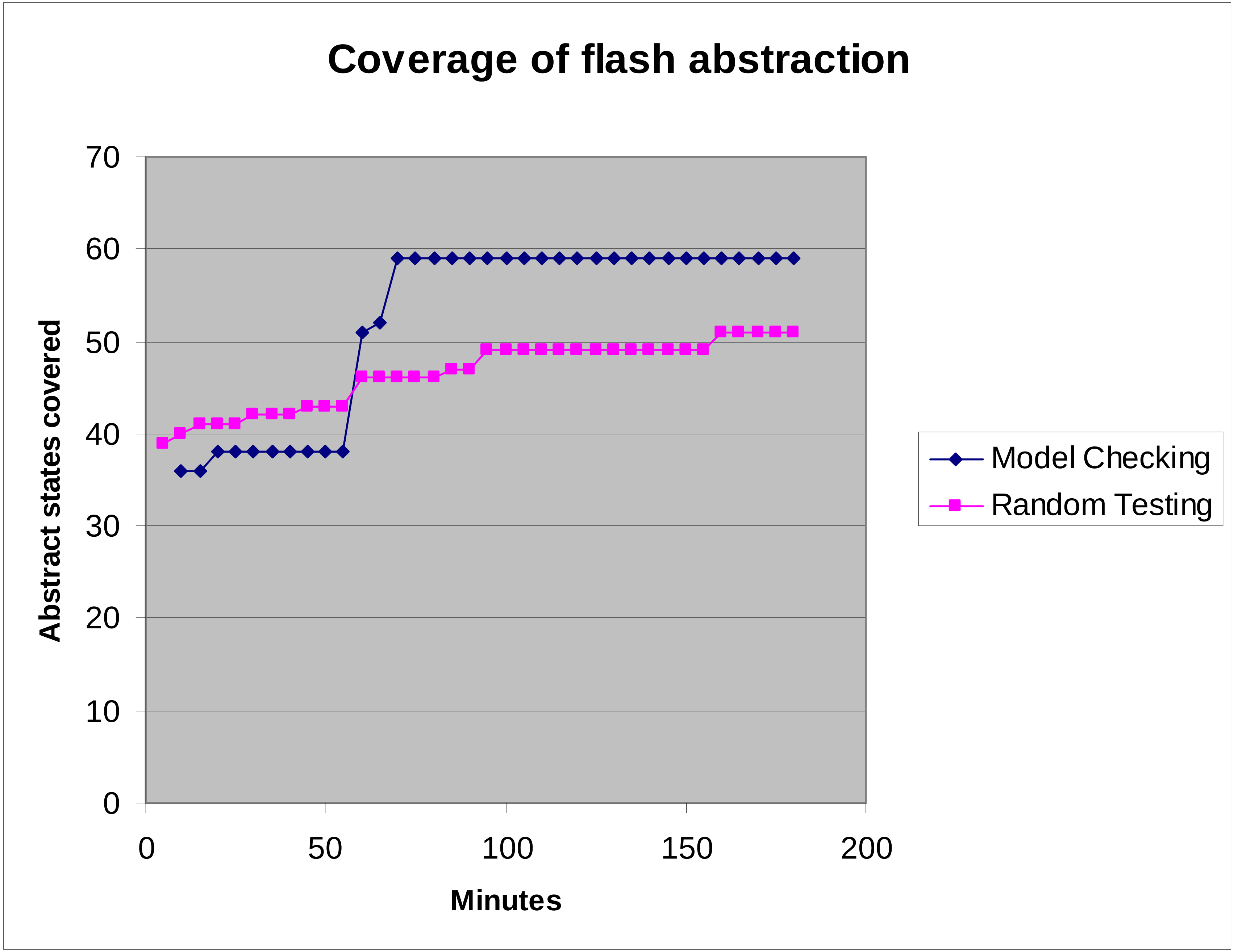
- From a flash file system for the Mars Science Laboratory mission – see the paper for details

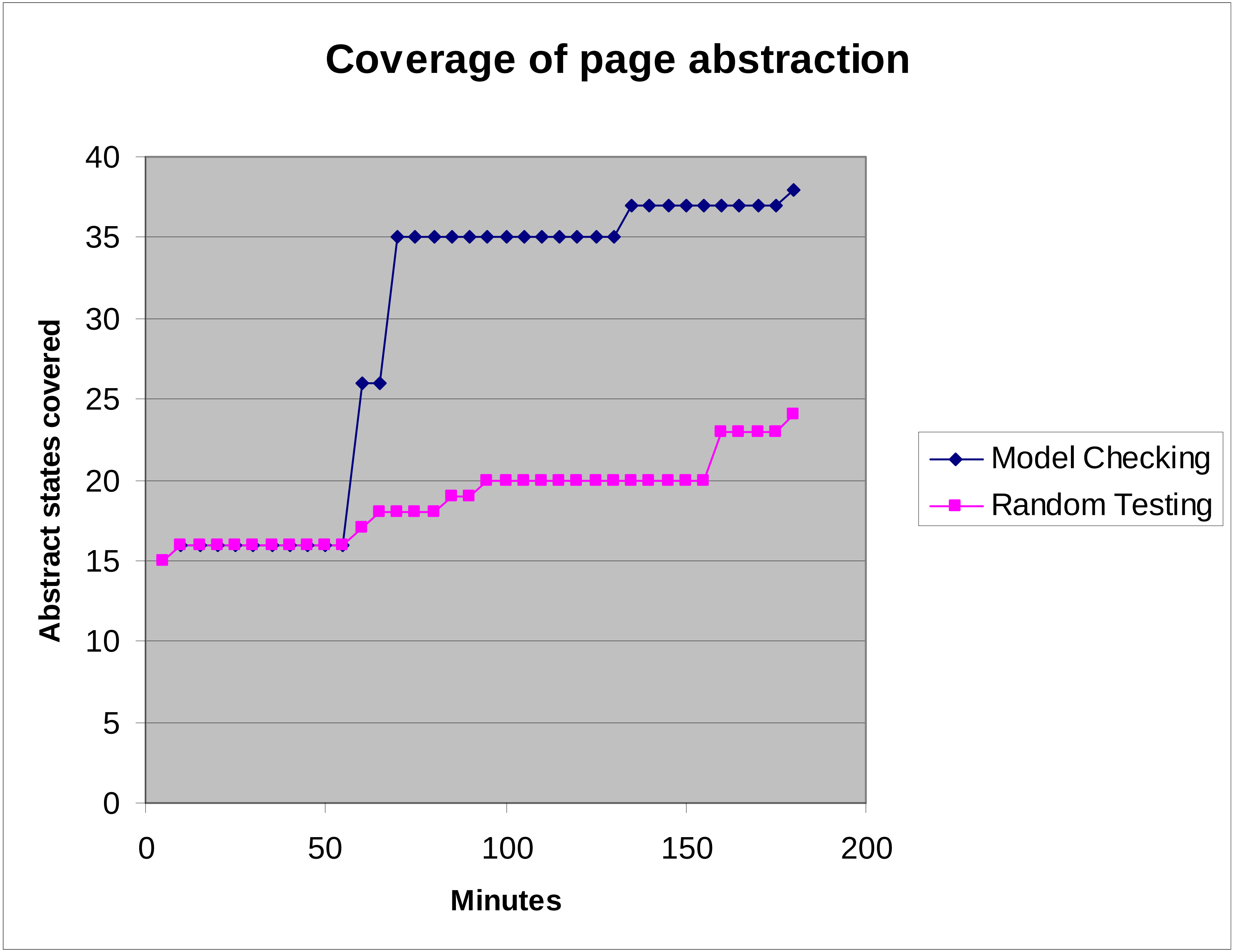
- Basic idea – how does coverage (source code / configurations of the flash file system) change as we increase testing time?

Coverage of nvds_box.c




Coverage of nvfs_pub.c
% Coverage
75.55
75.5
75.45
75.4
75.35
75.3
75.25
75.2
0
50
100
150
200
Minutes
Model Checking
Random Testing


Coverage of flash abstraction
Abstract states covered
0
10
20
30
40
50
60
70
Minutes
0
50
100
150
200
Model Checking
Random Testing


Coverage of page abstraction
Abstract states covered
40
35
30
25
20
15
10
5
0
0
50
100
150
200
Minutes
Model Checking
Random Testing

Conclusions (and an Invitation)

- Is model checking better?
  - Maybe, maybe not
  - Preliminary results for one program
  - Visser et al. and others report varying results for this question
  - These results don’t use as much feedback as our latest test harness – which may change the results (improves both model checking and random testing results)

Conclusions (and an Invitation)

- If you're analyzing or testing C programs
  - Where function-call level atomicity is ok
  - With well-defined memory usage
  - It might be well worth your while to try explicit-state model checking
  - Easy to work with abstractions and guide testing/analysis towards certain goals
  - Can also provide random testing "for free"
- JPF may work well for this purpose, also, though since it uses its own JVM, may be trickier/slower
- Download SPIN at http://www.spinroot.com