Software Property Checking

— Something like a Case Study —

Jan Tobias Mühlberg
<muehlber@cs.york.ac.uk>

York, March 9, 2006

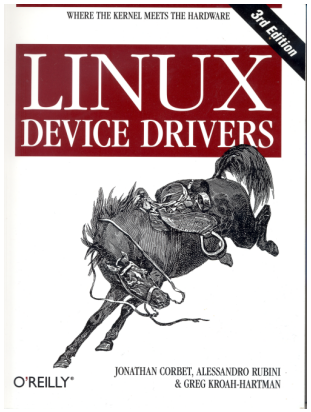
Motivation

- large scale software systems are difficult to build and usually contain bugs
- revealing, locating and correcting bugs is costly
- pure testing doesn't establish much trust in the code since it's never exhaustive
- verification would be nice but requires a specification. . .

Outline

1. Property Checking
2. BLAST – What is it?
3. Examples with BLAST
4. BLASTing the real world
5. Summary

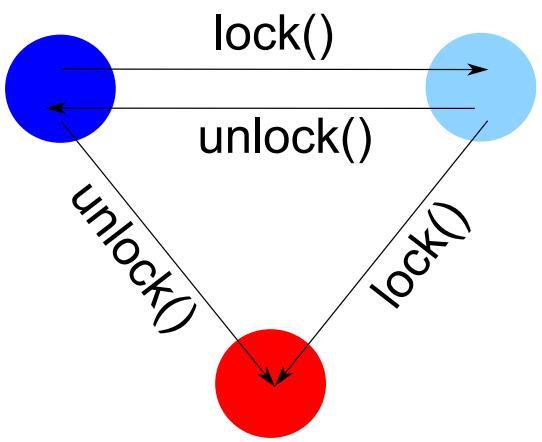
1. Property Checking



- use a *partial* specification of the program:

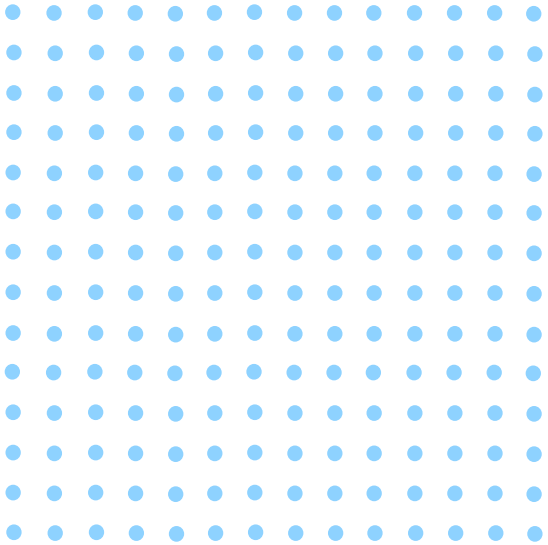
"Neither semaphores nor spinlocks allow a lock holder to acqire the lock a second time; should you attempt to do so, things simply hang." (Corbet et al., 2005)

1. Property Checking



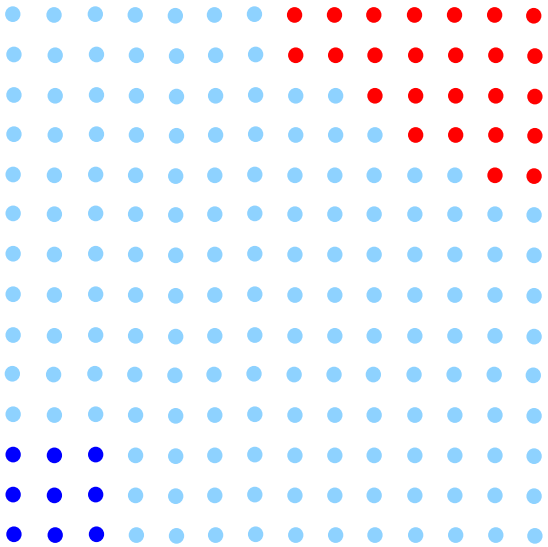
- use partial specification to identify invariants and error states
- then simply check whether the error states are reachable. . .

1. Property Checking



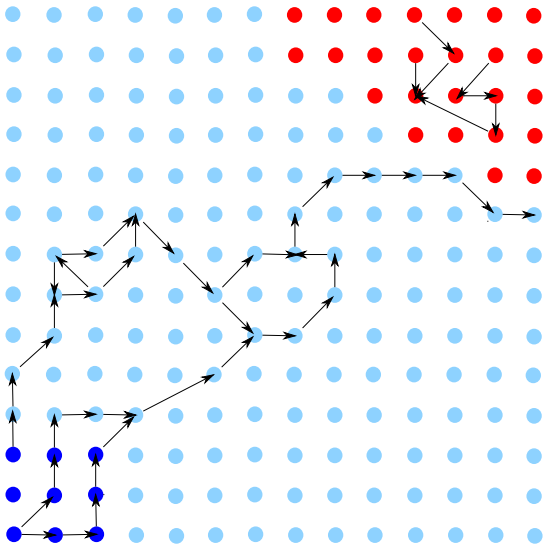
- our program consists of an infinite number of states

1. Property Checking



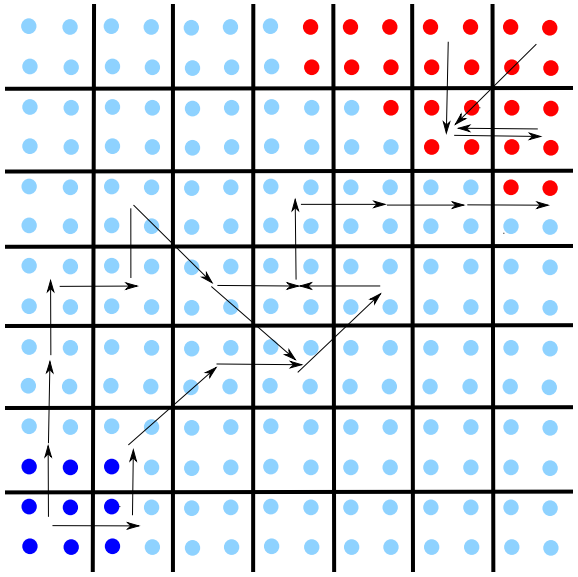
- some of them are initial states
- others are error states

1. Property Checking



- there are transitions connecting the states
- problem: Is there a path from an initial state to an error state?
- Undecidable.

1. Property Checking

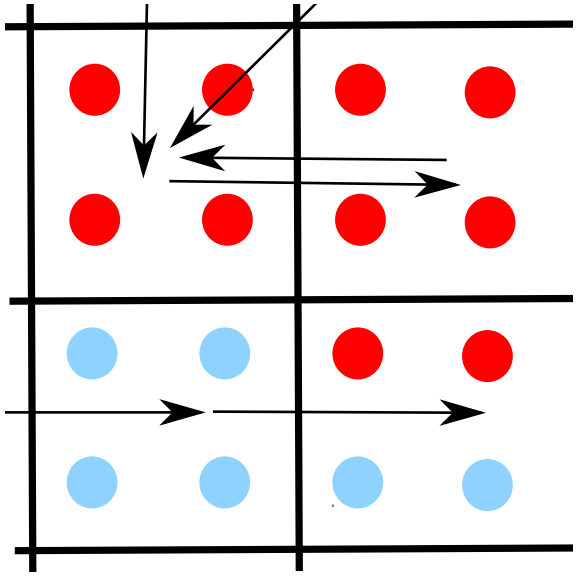


- states satisfying the same predicate are equivalent

$\longrightarrow$ can be merged into one abstract state

- number of abstract states is finite

1. Property Checking

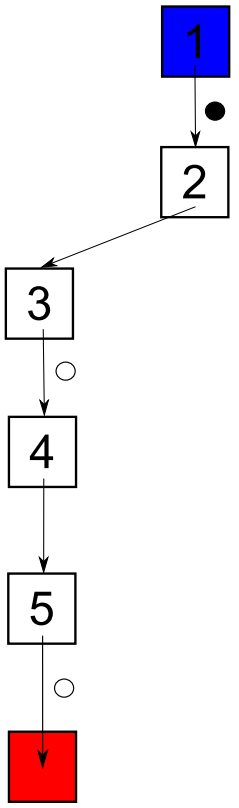


- over approximation:
  - no false negatives
  - false positives are common

→ abstraction needs to be refined

1. Property Checking

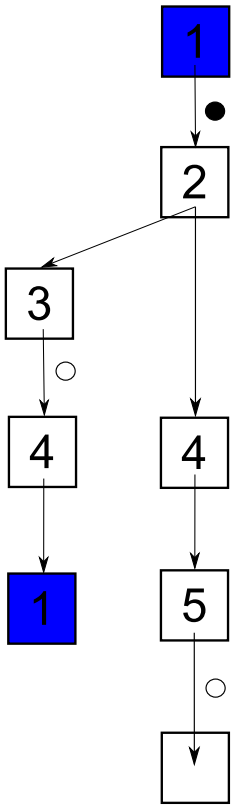
```
1  do {
     lock();
     old = new;
     q = q->next;
2    if ( q != NULL ) {
3      q->data = new;
       unlock();
       new++; }
4  } while (new != old);
5  unlock();
```



(Henzinger et al., 2005)

1. Property Checking

```
1  do {
     lock();
     old = new;
     q = q->next;
2    if ( q != NULL ) {
3      q->data = new;
       unlock();
       new++; }
4  } while (new != old);
5  unlock();
```



(Henzinger et al., 2005)

2. BLAST

- BLAST
  - generate instrumented program from C code; compute abstraction; model check the abstraction (Henzinger et al., 2002a)
  - has been applied to Windows and Linux device drivers (Henzinger et al., 2002a), (Beyer et al., 2005)
  - uses Lazy Abstraction algorithm. . . (Henzinger et al., 2002b)

2. BLAST

- Lazy Abstraction
  1. **abstraction**: build program as a finite push-down automaton; states represent truth assignments for predicates
  2. **verification**: reachability analysis
  3. **counterexample-driven refinement**: refine only error state
  4. goto 2

(Henzinger et al., 2002b)

3. Examples with BLAST (1)

```
input:
1  #include <assert.h>
2
3  int foo(int x, int y)
4  {
5   if (x > y)
6    {
7     x = y - x;
8     assert(x > 0);
9    } }
```

```
output:
The system is unsafe: example1.c
8 :: 8: FunctionCall(
 __assert_fail("x > 0", "ex0.c",
 8, "foo")) :: -1

8 :: 8: Pred(x@foo<=0) :: 8
7 :: 7: Block(x@foo = y@foo  -
 x@foo;) :: 8

5 :: 5: Pred(x@foo>y@foo) :: 7
```

3. Examples with BLAST (2)

```
input:
int main (void)
 {
  void *p1 = NULL;

  p1 = malloc (500);
  if ( p1 == NULL )
   { printf("out of memory.");
     return (1); }
  assert (p1 != NULL);
  free(p1); free(p1);
  return (0);
 }
```

```
spec:
global int allocstatus = 0;

event
 {pattern { $? = malloc($?); }
  action  { allocstatus = 1; } }

event
 {pattern { $? = free($?); }
  guard   { allocstatus == 1 }
  action  { allocstatus = 0; } }
```

4. BLASTing the real world

- the major problems in large scale software systems:
  - dynamic allocation, use and de-allocation of memory
  - locking, deadlock avoidance

(Chou et al., 2001)

4. BLASTing the real world

```
static int i2o_pci_probe (...)
{
 int rc;
 struct i2o_controller *c;
 ...
 c = i2o_iop_alloc();
 ...
 i2o_iop_free(c);
 ...
 put_device(c->device.parent);
 ...
 return rc;
}
```

- add status field to `i2o_controller`
- add error labels to `i2o_iop_free` and `put_device`
- then run the model checker...

4. BLASTing the real world

```
void i2o_iop_free
 (struct i2o_controller *c)
{/* was: kfree(c); */

 if (c->alloc_status == 0) {
  goto ERROR; }
 else {
  c->alloc_status = 0;
  return; }

 ERROR: return;
}
```

- . . . and see how it fails.

$\longrightarrow$ correlate pointers and structs

$\longrightarrow$ aliasing

$\longrightarrow$ function pointers

$\longrightarrow$ global locks

2. BLAST

- correlate pointers and structs
  - most pointer operations are ignored,
    even the &-operator is not covered
  - objects behind a pointer is not passed
    through function calls
  - functions not available in source code
    are assumed not to affect variables

2. BLAST

- function pointers

```
struct i2o_controller *
 i2o_iop_alloc (void)
{ struct i2o_controller *c;
  ...
  c = kmalloc(sizeof(*c), ...);
  ...
  c->device.release =
   &i2o_iop_release;
  ...
  return c; }
```

- structures often contain "destructors"
- calling `c->device.release()` is equivalent to `i2o_iop_free(c)`

2. BLAST

- global locks
  - most locking problems do not result form local locks but from ineraction with locks in other components
  - functions that are not available in source code are assumed to have no effect. . .
  - BLAST doesn't deal well with concurrent execution paths

5. Summary

- BLAST is not suitable for verifying complex properties like absence of deadlocks and memory safety on large scale software systems.

Thank you!

References

Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R.: 2005, Checking memory safety with blast, in M.Cerioli (ed.), *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005), Edinburgh, April 2-10, volume 3442 of Lecture Notes in Computer Science*, pp 2 – 18, Springer-Verlag, Berlin, Germany

Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R.: 2001, An empirical study of operating system errors, in *Symposium on Operating Systems Principles*, pp 73 – 88

Corbet, J., Rubini, A., and Kroah-Hartmann, G.: 2005, *Linux Device Drivers*, O'Reilly, Sebastopol, CA, USA, 3rd edition

Henzinger, T., Jhala, R., and Majumdar, R.: 2005, *The BLAST Software Verification System*, Tutorial at the 12th SPIN Workshop on Model Checking Software

Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W.: 2002a, Temporal-safety proofs for systems code, in *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pp 526 – 538, Springer-Verlag, London, UK

Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G.: 2002b, Lazy abstraction, in *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Portland, Oregon*, pp 58 – 70, ACM Press, New York, NY, USA