CS 267: Automated Verification

Lecture 11: Software Model Checking

Instructor: Tevfik Bultan

Software's Chronic Crisis

Large software systems often:

- Do not provide the desired functionality
- Take too long to build
- Cost too much to build
- Require too much resources (time, space) to run
- Cannot evolve to meet changing needs

  - For every 6 large software projects that become operational, 2 of them are canceled
  - On the average software development projects overshoot their schedule by half
  - 3 quarters of the large systems do not provide required functionality

Software Failures

- There is a long list of failed software projects and software failures

- You can find a list of famous software bugs at:
  http://www5.in.tum.de/~huckle/bugse.html

- I will talk about two famous and interesting software bugs

Ariane 5 Failure

- A software bug caused European Space Agency's Ariane 5 rocket to crash 40 seconds into its first flight in 1996 (***cost: half billion dollars***)

- The bug was caused because of a software component that was being reused from Ariane 4
- A software exception occurred during execution of a data conversion from 64-bit floating point to 16-bit signed integer value
  - The value was larger than 32,767, the largest integer storable in a 16 bit signed integer, and thus the conversion failed and an exception was raised by the program
- When the primary computer system failed due to this problem, the secondary system started running.
  - The secondary system was running the same software, so it failed too!

Ariane 5 Failure

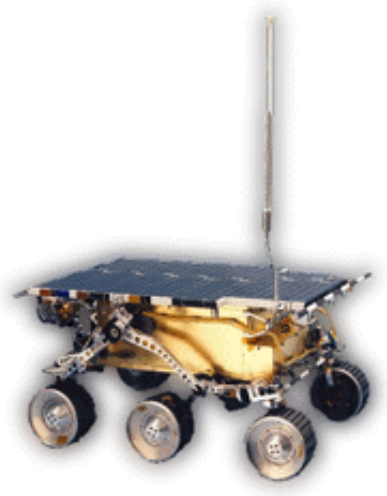
- The programmers for Ariane 4 had decided that this particular velocity figure would never be large enough to raise this exception.
  - Ariane 5 was a faster rocket than Ariane 4!
- The calculation containing the bug actually served no purpose once the rocket was in the air.
  - Engineers chose long ago, in an earlier version of the Ariane rocket, to leave this function running for the first 40 seconds of flight to make it easy to restart the system in the event of a brief hold in the countdown.

- You can read the report of Ariane 5 failure at:

  http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html

Mars Pathfinder

- A few days into its mission, NASA's Mars Pathfinder computer system started rebooting itself
  - Cause: Priority inversion during preemptive priority scheduling of threads



- Priority inversion occurs when
  - a thread that has higher priority is waiting for a resource held by thread with a lower priority
- Pathfinder contained a data bus shared among multiple threads and protected by a mutex lock
- Two threads that accessed the data bus were: a high-priority bus management thread and a low-priority meteorological data gathering thread
- Yet another thread with medium-priority was a long running communications thread (which did not access the data bus)

Mars Pathfinder

- The scenario that caused the reboot was:
  - The meteorological data gathering thread accesses the bus and obtains the mutex lock
  - While the meteorological data gathering thread is accessing the bus, an interrupt causes the high-priority bus management thread to be scheduled
  - Bus management thread tries to access the bus and blocks on the mutex lock
  - Scheduler starts running the meteorological thread again
  - Before the meteorological thread finishes its task yet another interrupt occurs and the medium-priority (and long running) communications thread gets scheduled
  - At this point high-priority bus management thread is waiting for the low-priority meteorological data gathering thread, and the low-priority meteorological data gathering thread is waiting for the medium-priority communications thread
  - Since communications thread had long-running tasks, after a while a watchdog timer would go off and notice that the high-priority bus management thread has not been executed for some time and conclude that something was wrong and reboot the system

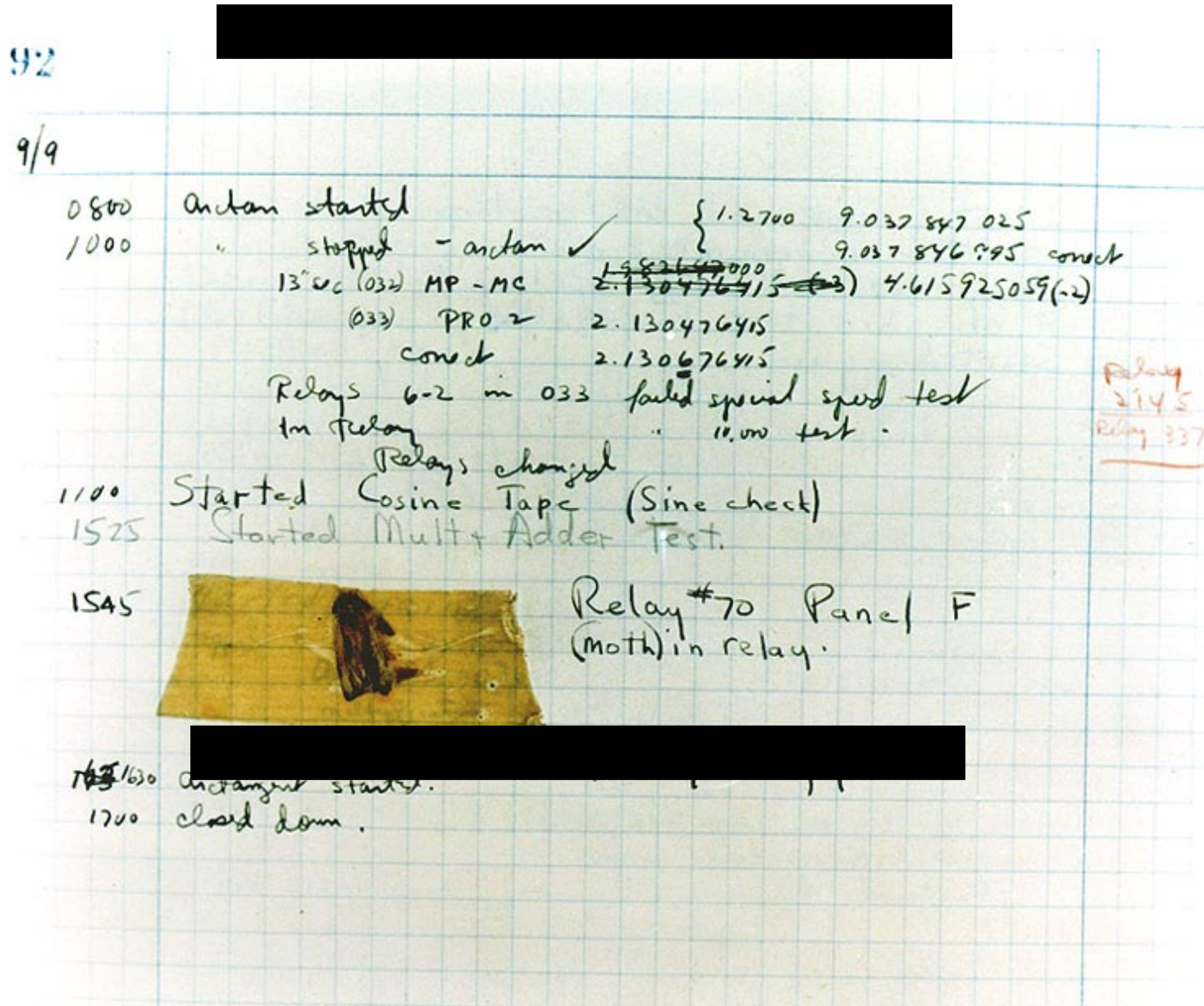
Software's Chronic Crisis

- Software product size is increasing exponentially
  - faster, smaller, cheaper hardware
- Software is everywhere: from TV sets to cell-phones
- Software is in safety-critical systems
  - cars, airplanes, nuclear-power plants
- We are seeing more of
  - distributed systems
  - embedded systems
  - real-time systems
    - These kinds of systems are harder to build
- Software requirements change
  - software evolves rather than being built

Summary

- Software's chronic crisis: Development of large software systems is a challenging task
  - Large software systems often: Do not provide the desired functionality; Take too long to build; Cost too much to build Require too much resources (time, space) to run; Cannot evolve to meet changing needs

What is this?


92
[redacted]
9/9
0800 Arctan started
1000 " stopped - arctan ✓
1.2700 9.037 847 025
9.037 846 995 conect
13" uc (032) MP - MC
4.615925059(-2)
(033) PRO 2 2.130476415
conect 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult+ Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
[redacted]
1700 closed down.

First Computer Bug

- In 1947, Grace Murray Hopper was working on the Harvard University Mark II Aiken Relay Calculator (a primitive computer).
- On the 9th of September, 1947, when the machine was experiencing problems, an investigation showed that there was a moth trapped between the points of Relay #70, in Panel F.
- The operators removed the moth and affixed it to the log. The entry reads: "First actual case of bug being found."
- The word went out that they had "debugged" the machine and the term "debugging a computer program" was born.

Can Model Checking Help

- The question is: Can the automated verification techniques we have been discussing be used in finding bugs in software systems?

- Today I will discuss some tools that have been successful in identifying bugs in software

Model Checking Evolution

- Earlier model checkers had their own input specification languages
  - For example Spin, SMV
- This requires translation of the system to be verified to the input langauge of the model checker
  - Most of the time these translations are not automated and use ad-hoc simplifications and abstractions
- More recently several researchers developed tools for model checking programs
  - These model checkers work directly on programs, i.e., their input language is a programming language
  - These model checkers use well-defined techniques for restricting the state space or use automated abstraction techniques

Explicit-State Model Checking Programs

- Verisoft from Bell Labs
  - C programs, handles concurrency, bounded search, bounded recursion.
  - Uses stateless search and partial order reduction.

- Java Path Finder (JPF) at NASA Ames
  - Explicit state model checking for Java programs, bounded search, bounded recursion, handles concurrency.
  - Uses techniques similar to the techniques used in Spin.

- CMC from Stanford for checking systems code written in C
  - In terms of model checking techniques it is similar to JPF

Symbolic Model Checking of Programs

- CBMC
  - This a bounded model checker that bounds the loop iterations and recursion depth.
  - Uses a SAT solver.

- SLAM project at Microsoft Research
  - Symbolic model checking for C programs. Can handle unbounded recursion but does not handle concurrency.
  - Uses predicate abstraction and BDDs.

Java Path Finder

- Program checker for Java
- Properties to be verified
  - Properties can be specified as assertions
    - static checking of assertions
  - It can also verify LTL properties
- Implements both depth-first and breadth-first search and looks for assertion violations statically
- Uses static analysis techniques to improve the efficiency of the search
- Requires a complete Java program
- It can only handle pure Java, it cannot handle native code

Java Path Finder, First Version

- First version
  - A translator from Java to PROMELA
  - Use SPIN for model checking

- Since SPIN cannot handle unbounded data
  - Restrict the program to finite domains
    - A fixed number of objects from each class
    - Fixed bounds for array sizes

- Does not scale well if these fixed bounds are increased

- Java source code is required for translation

Java Path Finder, Second Version

- Later version of the JPF has its own virtual machine: JPF-JVM
  - Executes Java bytecode
    - can handle pure Java but can not handle native code
  - Has its own garbage collection
  - Stores the visited states and stores current path
  - Offers some methods to the user to optimize verification
- Traversal algorithm
  - Traverses the state-graph of the program
  - Tells JPF-JVM to move forward, backward in the state space, and evaluate the assertion

- The rest of the slides are on later JPF version discussed in:

W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. "Model Checking Programs." Automated Software Engineering Journal Volume 10, Number 2, April 2003.

Storing the States

- JPF implements a depth-first search on the state space of the given Java program
  - To do depth first search we need to store the visited states
    - There are also verification tools which use stateless search such as Verisoft
- The state of the program consists of
  - information for each thread in the Java program
    - a stack of frames, one for each method called
  - the static variables in classes
    - locks and fields for the classes
  - the dynamic variables (fields) in objects
    - locks and fields for the objects

Storing States Efficiently

- Since different states can have common parts each state is divided to a set of components which are stored separately
  - locks, frames, fields
- Keep a pool for each component
  - A table of field values, lock values, frame values
- Instead of storing the value of a component in a state, store an index in the state denoting where the component is stored in the table
  - The whole state becomes an integer vector
- JPF collapses states to integer vectors using this idea

State Space Explosion

- State space explosion is one of the major challenges in model checking
- The idea is to reduce the number of states that have to be visited during state space exploration
- Here are some approaches used to attack state space explosion
  - Symmetry reduction
    - search equivalent states only once
  - Partial order reduction
    - do not search thread interleavings that generate equivalent behavior
  - Abstraction
    - Abstract parts of the state to reduce the size of the state space

Symmetry Reduction

- Some states of the program may be equivalent
  - Equivalent states should be searched only once
- Some states may differ only in their memory layout, the order objects are created, etc.
  - these may not have any effect on the behavior of the program
- JPF makes sure that the order which the classes are loaded does not effect the state
  - There is a canonical ordering of the classes in the memory

Symmetry Reduction

- A similar problem occurs for location of dynamically allocated objects in the heap
  - If we store the memory location as the state, then we can miss equivalent states which have different memory layouts
  - JPF tries to remedy this problem by storing some information about the `new` statements that create an object and the number of times they are executed

Partial Order Reduction

- Statements of concurrently executing threads can generate many different interleavings
  - all these different interleavings are allowable behavior of the program
- A model checker has to check all possible interleavings that the behavior of the program is correct in all cases
  - However different interleavings may generate equivalent behaviors
- In such cases it is sufficient to check just one interleaving without exhausting all the possibilities
  - This is called partial order reduction

state space search generates 258 states
with symmetry reduction: 105 states
with partial order reduction: 68 states
with symmetry reduction + partial order reduction : 38 states

```
class S1 { int x;}
class FirstTask
  extends Thread {
  public void run() {
  S1 s1; int x = 1;
  s1 = new S1();
  x = 3;
}}
```

```
class S2 { int y;}
class SecondTask
  extends Thread {
  public void run() {
  S2 s2; int x = 1;
  s2 = new S2();
  x = 3;
}}
```

```
class Main {
  public static void main(String[] args) {
    FirstTask task1 = new FirstTask();
    SecondTask task2 = new SecondTask();
    task1.statr(); task2.start();
}}
```

Static Analysis

- JPF uses following static analysis techniques for reducing the state space:
  - slicing
  - partial evaluation
- Given a slicing criterion slicing reduces the size of a program by removing the parts of the program that have no effect on the slicing criterion
  - A slicing criterion could be a program point
  - Program slices are computed using dependency analysis
- Partial evaluation propagates constant values and simplifies expressions

Abstraction vs. Restriction

- JPF also uses abstraction techniques such as predicate abstraction to reduce the state space
- Still, in order to check a program with JPF, typically, you need to restrict the domains of the variables, the sizes of the arrays, etc.
- Abstraction over approximates the program behavior
  - causes spurious counter-examples
- Restriction under approximates the program behavior
  - may result in missed errors
- If both under and over approximation techniques are used then the resulting verification technique is neither sound nor complete
  - However, it is still useful as a debugging tool and it is helpful in finding bugs

JPF Java Modeling Primitives

- Atomicity (used to reduce the state space)
  - `beginAtomic(), endAtomic()`

- Nondeterminism (used to model non-determinism caused by abstraction)
  -
```
int random(int);
boolean randomBool();
Object randomObject(String cname);
```

- Properties (used to specify properties to be verified)
  - `AssertTrue(boolean cond)`

Annotated Java Code for a Reader-Writer Lock

```
import gov.nasa.arc.ase.jpf.jvm.Verify;
class ReaderWriter {
private int nr;
private boolean busy;
private Object Condr_enter;
private Object Condw_enter;
public ReaderWriter() {
  Verify.beginAtomic();
  nr = 0; busy=false ;
  Condr_enter =new Object();
  Condw_enter =new Object();
  Verify.endAtomic();
}
public boolean read_exit(){
  boolean result=false;
  synchronized(this){
    nr = (nr - 1);
    result=true;
  }
  Verify.assertTrue(!busy || nr==0 );
  return result;
}
```

```
private boolean Guarded_r_enter(){
  boolean result=false;
  synchronized(this){
  if(!busy){nr = (nr +
   1);result=true;}}
  return result;
}
public void read_enter(){
  synchronized(Condr_enter){
  while (! Guarded_r_enter()){
    try{Condr_enter.wait();}
    catch(InterruptedException e){}
  }}
  Verify.assertTrue(!busy || nr==0 );
}
private boolean Guarded_w_enter(){…}
public void write_enter(){…}
public boolean write_exit(){…}
};
```

JPF Output

```
>java gov.nasa.arc.ase.jpf.jvm.Main rwmain

JPF 2.1 - (C) 1999,2002 RIACS/NASA Ames Research Center
JVM 2.1 - (C) 1999,2002 RIACS/NASA Ames Research Center

Loading class gov.nasa.arc.ase.jpf.jvm.reflection.JavaLangObjectReflection
Loading class gov.nasa.arc.ase.jpf.jvm.reflection.JavaLangThreadReflection
==============================
  No Errors Found
==============================

----------------------------------
States visited       : 36,999
Transitions executed : 68,759
Instructions executed: 213,462
Maximum stack depth  : 9,010
Intermediate steps   : 2,774
Memory used          : 22.1MB
Memory used after gc : 14.49MB
Storage memory       : 7.33MB
Collected objects    : 51
Mark and sweep runs  : 55,302
Execution time       : 20.401s
Speed                : 3,370tr/s
----------------------------------
```

Example Error Trace

```
1 error found: Deadlock
=======================
 *** Path to error: ***
=======================

Steps to error: 2521
Step #0 Thread #0
Step #1 Thread #0
  rwmain.java:4         ReaderWriter monitor=new  ReaderWriter();
Step #2 Thread #0
  ReaderWriter.java:10          public ReaderWriter( ) {
…
Step #2519 Thread #2
  ReaderWriter.java:71            while (! Guarded_w_enter()){
Step #2520 Thread #2
  ReaderWriter.java:73                              Condw_enter.wait();
```

Model Checking System Code

- “CMC: A pragmatic approach to model checking real code,”
  Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, David L. Dill. Operating System Design and Implementation (OSDI) 2002.

- “Using Model Checking to Find Serious File System Errors,”
  Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi, Operating System Design and Implementation (OSDI) 2004.

CMC

- C Model Checker (CMC)
- A tool for model checking C code
  - Explicit state model checker
  - Uses on-the-fly state space exploration
  - Uses depth first search
- Used for verification of AODV protocol implementations
  - Found bugs in the implementations
  - Found a bug in the AODV specification

CMC

- A CMC model consists of a set of concurrently executing processes
  - The execution of the system corresponds to the interleaved execution of transitions (atomic steps) of the processes
- CMC was developed to target event-driven applications
  - A set of event-handler routines process incoming events such as packet arrivals
- In CMC each transition corresponds to execution of an event-handler
  - Execution of an event-handler is considered to an atomic step of execution

Model Checking Event Driven Software

- In order to check an event-driven application with CMC the user has to do the following:
  - Specify Correctness properties
    - Some generic properties such as illegal memory access, memory leakage can be checked for all applications
    - Application specific properties can be inserted as assertions at different program points
  - Specify the environment
    - Write stubs for functions
    - Model the network as a simple queue
    - To represent non-determinism use CMCChoose
  - Determine the initialization functions and the event-handlers

Model Checking with CMC

- CMC calls the initialization functions to compute the initial state
- CMC generates the state graph on the fly by computing the successors states of the state that is on the top of the stack
  - There are two reasons why a state may have more than one successor
    - Nondeterminism that is introduced due to concurrency, i.e., different interleavings of concurrent processes
    - Nondeterminism that is explicitly introduced during environment modeling

Model Checking with CMC

- In order to check for memory errors CMC does the following
  - Randomly overwrite freed memory
  - Call memory clean-up functions to free all the memory. If there is any memory left allocated, call it as leaked

- CMC also checks for violations of user specified assertions during its execution

Hash Compaction

- CMC reduces the state space by not storing all the state information for the visited states
  - For each state CMC computes a small signature (4 to 8 bytes). Instead of storing the whole state, this small signature is stored in the hash table VisitedStates
  - If there is a hash conflict, we can ignore a visited state during stat exploration by thinking that it was visited before even if it was not visited before
  - Note that this is similar to bit-state hashing and can cause us to miss errors, i.e., it is not sound!
  - The counter-example paths that are reported will not be spurious since the Stack keeps the exact state information (which is also necessary for computing the next states)

Memory Layout, Data Structures

- Two equivalent states may be marked as different if they have different memory layouts
- CMC supports traversals of the data structures to transform them, so that they are stored based on their traversal order.
  - If the traversal gives a deterministic order, then equivalent states will end up having equivalent memory layouts.

Memory Layout, Data Structures

- Also state of data structures may significantly increase the size of the state space
  - For example an unordered collection of objects has many possible configurations
  - If the behavior of the program is independent of the order of objects in the list, then storing the list as an ordered list would reduce the state space
    - since a large number of unordered collections will be considered equivalent to an ordered collection

- These correspond to the symmetry reductions that I mentioned when I was discussing JPF

Bounded Search

- In cases where the tool is unable to scale, they reduce the sizes of the models
  - For example reduce the number of routing nodes
- They call this down-scaling
- This may also lead to missed errors

Checking AODV

- CMC was used to check three AODV implementations
  - mad-hoc, Kernel AODV, AODV-UU: 3-5K lines of code
- Correctness specification

  ~300 lines of code
- Environment
  - Network environment model: 400 lines of code
  - Other stubs: 100-200 lines of code
- State Canonicalization (i.e., symmetry reduction)

  ~200 lines of code
- Bounded search
  - Restricted the protocol to run with 2 to 4 processes
  - Bounded the message queue size to 1 to 3

Bugs in AODV implementations

- Memory errors
  - Not checking for allocation failure: 12
  - Not freeing allocated memory: 8
  - Using memory after freeing it: 2
- Unexpected messages: 2
- Invalid messages
  - Invalid packets: 4
  - Uninitialized variables: 2
  - Integer overflow: 2
- Routing loops
  - Implementation error: 2
  - Specification error: 1
- About 1 bug per 300 lines of code

Model Checking File Systems

- FiSC: A model checking tool for file systems
  - Built on CMC

- FiSC was applied to three widely-used, heavily-tested file systems:
  - ext3, JFS, and ReiserFS.
  - They found serious bugs in all of them, 32 in total.
  - Most have led to patches within a day of diagnosis.
  - For each file system, FiSC found demonstrable events leading to the unrecoverable destruction of metadata and entire directories, including the file system root directory ″/″.

Model Checking File Systems

- It is hard to test file systems
  - Testing that a file system correctly recovers from a crash requires a crash-reboot-reconstruct cycle which takes about a minute per test case
  - Testing all crash possibilities is impossible

- Model checking can be used to systematically explore the state space of the file system

FiSC

- FiSC contains four components
  - CMC model checker running the Linux kernel
  - A file system test driver
  - A permutation checker which verifies that the file system can recover no matter what order the buffer cache contents are written to disk
  - A fsck recovery checker that checks failures during recovery are handled correctly

- FiSC provides a block device driver layer using the Virtual File System (VFS) interface

Checks

- Generic checks that apply to any code run in the kernel
  - deadlock, null dereferences, call sequencing for paired functions, memory leak, no silent failures
- Consistency checks
  - a system call that returns a failure should not result in a change
  - different write orderings from the buffer to disk should recover to a valid state
  - If a block is changed in the buffer cache, it should be marked as dirty
  - buffer consistency: each file system has a set of rules about how the buffers should be marked
  - Running fsck in default or comprehensive mode should generate the same result

Checking Overview

- The model checker starts with an empty and formatted disk (initial state)

- Recursively generates successive states by executing transitions

Checking Overview

- Transitions are either test driver operations or FS-specific kernel threads which flush blocks to disk
  - The test driver executes possible operations
    - Creates, removes and renames files, directories
    - Writes to and truncates files
    - Mounts and unmounts the file system

- As each state is generated, all disk writes by the file system are intercepted and forwarded to the permutation checker
  - Permutation checker checks that the disk is in a state that fsck can repair to produce a valid file system after each subset of all possible disk writes

Recovery Checking

- They simulate a crash and they create two scenarios from the crash state
  - In one scenario fsck is run to completion
  - In the other scenario they simulate another crash during the execution of fsck and run fsck one more time to recover
  - These two scenarios should lead to identical disk states

Model Checking File Systems

- FiSC was applied to three widely-used, heavily-tested file systems:
  - ext3, JFS, and ReiserFS.

- They found serious bugs in all of them, 32 in total.
  - Most have led to patches within a day of diagnosis.
  - For each file system, FiSC found demonstrable events leading to the unrecoverable destruction of metadata and entire directories, including the file system root directory ``/''.