

---

# Dynamic Software Model Checking

Patrice Godefroid

Microsoft Research

# Ed Clarke: A man, An idea...

---

- LASER'2011 summer school (Elba island, Italy)



# Ed Clarke: A man, An idea...

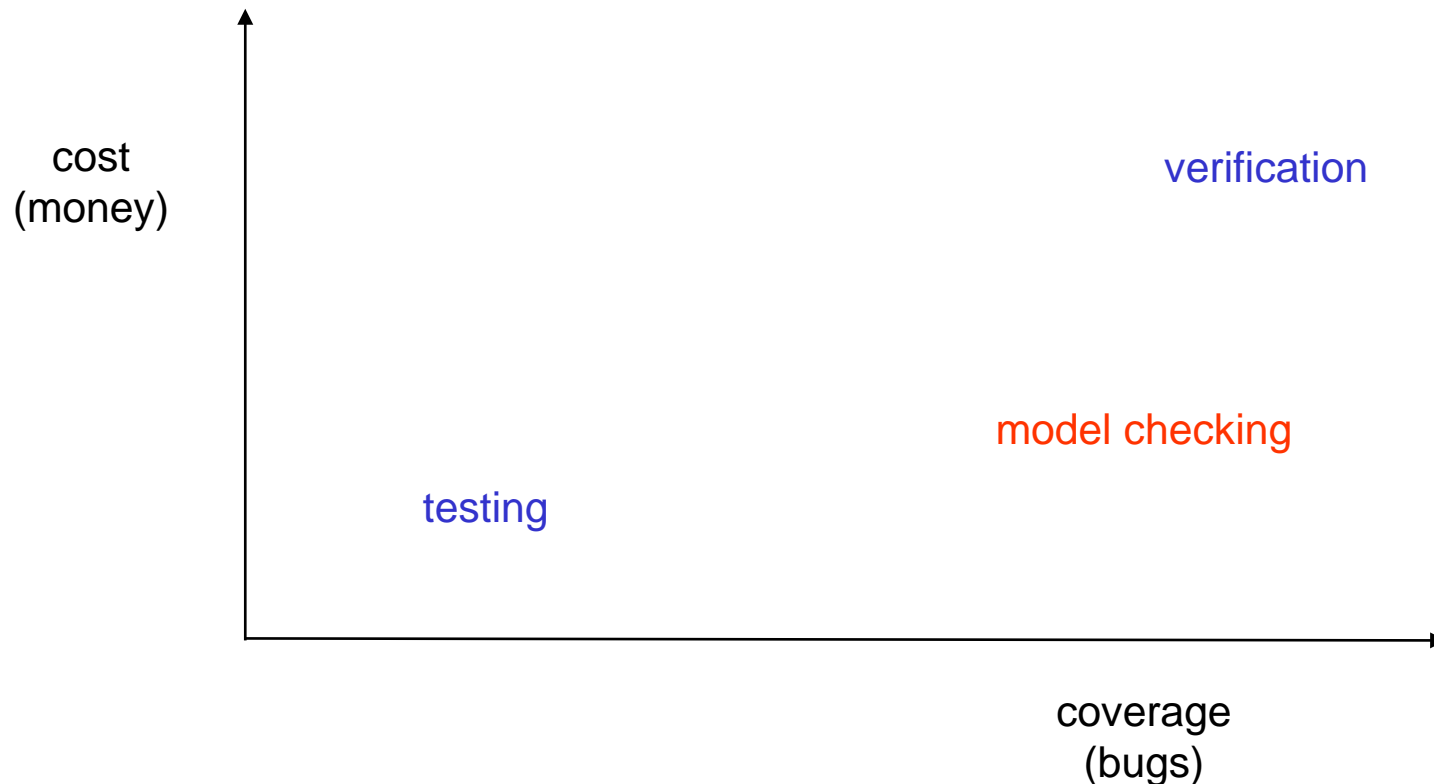
---

- LASER'2011 summer school (Elba island, Italy)
- Q from student: "career advice for young researcher?"
- Ed: "Pick an idea that excites you, then devote your life to it."



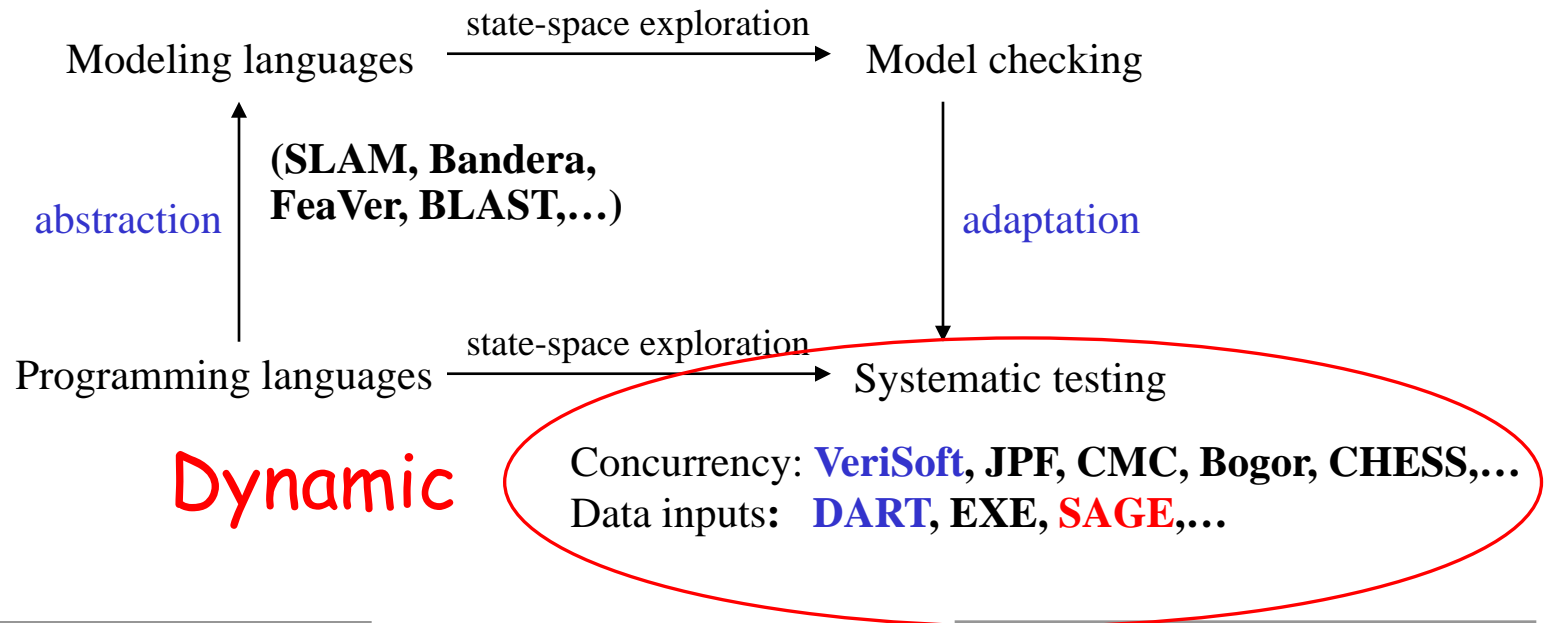
# Insight: Model Checking is **Super Testing**

- Simple yet effective technique for **finding bugs**
- In the software-engineering universe:



# Dynamic Software Model Checking

- How to apply model checking to analyze **software**?
  - “Real” programming languages (e.g., C, C++, Java),
  - “Real” size (e.g., 100,000’s lines of code).
- Two main approaches to software model checking:



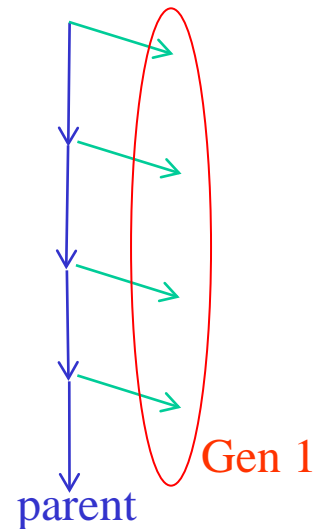
# Example: SAGE @ Microsoft

- Problem: How to systematically explore efficiently the **state spaces** of **sequential programs** to find bugs due to **malformed inputs**?
- Motivation: security testing at Microsoft
- Software security bugs can be very expensive:
  - Cost of each Microsoft Security Bulletin: \$Millions
  - Cost due to worms (Slammer, CodeRed, Blaster, etc.): \$Billions
- Many security exploits are initiated via files or packets
  - Ex: MS Windows includes parsers for hundreds of file formats
- Security testing: "hunting for **million-dollar bugs**"



# A Solution: Whitebox Fuzzing [NDSS'08]

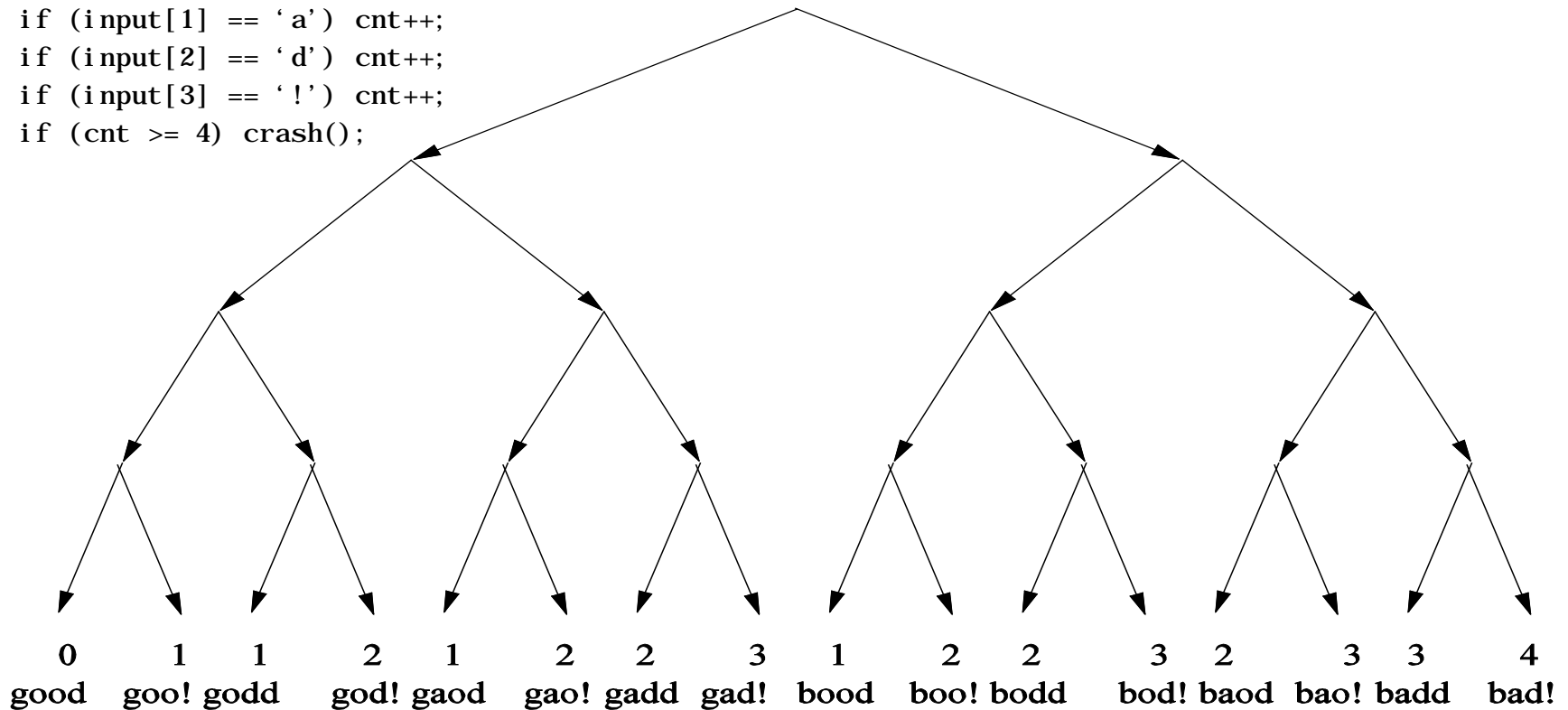
- Idea: mix fuzz testing with **dynamic test generation**
  - **Dynamic symbolic execution** to collect constraints on inputs
  - Negate those, **solve new constraints** to get new tests
  - Repeat → "systematic dynamic test generation" (= **DART**)  
( Why **dynamic** ? Because **most precise** ! [PLDI'05, PLDI'11] )
- Combine with a **generational** search (not DFS)
  - Negate 1-by-1 **each** constraint in a path constraint
  - Generate **many** children for each parent run
  - Challenge **all** the layers of the application sooner
  - Leverage expensive symbolic execution
- Implemented in the tool **SAGE**
  - Optimized for **large** x86 trace analysis, file fuzzing



# The Search Space

```
void top(char input[4])
{
  int cnt = 0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 4) crash();
}
```

If symbolic execution is perfect  
and search space is small,  
this is **verification!**

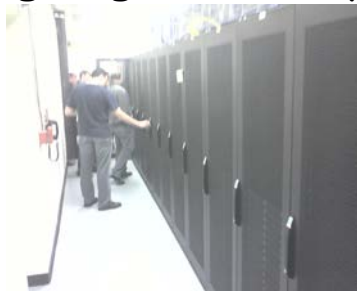




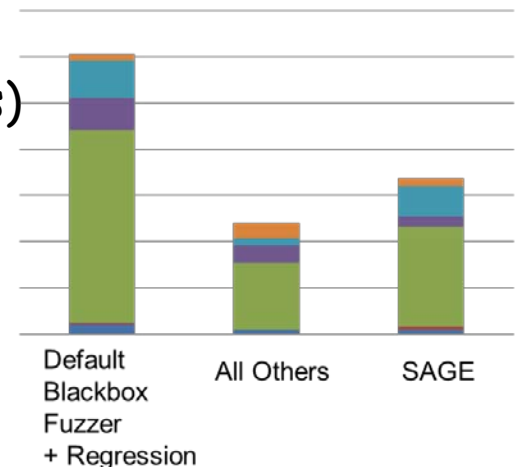
# SAGE Results

Since 2007: many new security bugs found  
(missed by blackbox fuzzers, static analysis)

- Apps: image decoders, media players, document processors,...
- Bugs: Write A/Vs, Read A/Vs, Crashes,...
- Many triaged as "security critical, severity 1, priority 1"  
(would trigger Microsoft security bulletin if known outside MS)
- Example: WEX Security team for Win7
  - Dedicated fuzzing lab with 100s machines
  - 100s apps (deployed on 1 billion+ computers)
  - ~1/3 of **all** fuzzing bugs found by SAGE !



How fuzzing bugs found (2006-2009) :



# Impact of SAGE (in Numbers)

---

- 500+ machine-years
  - Runs in the largest dedicated fuzzing lab in the world
  - Largest computational usage ever for any SMT solver
- 100s of apps, 100s of bugs (missed by everything else)
  - Bug fixes shipped quietly (no MSRCs) to 1 Billion+ PCs
  - Millions of dollars saved (for Microsoft and the world)
- "Practical Verification":
  - Eradicate **all** buffer overflows in **all** Windows parsers
    - <5 security bulletins in all SAGE-cleaned Win7 parsers, 0 since 2011
    - If nobody can find bugs in P, P is observationally equiv to "verified"!
    - Reduce costs & risks for Microsoft, increase those for Black Hats

2000

2005

2010

2015

---

Blackbox Fuzzing

Whitebox Fuzzing

---

"Practical Verification"

# Conclusion: Ed Clarke

---

- A man
- An idea
- A community
- Changing the world



(Elba, 2011)

**Thank you !**

*There is one thing stronger than all the armies in the world;  
and that is an idea whose time has come. -- Victor Hugo*

# Example

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 4) crash();
}
```

input = "good"

Path constraint:

$I_0 \neq \text{'b'}$  →  $I_0 = \text{'b'}$

$I_1 \neq \text{'a'}$  →  $I_1 = \text{'a'}$

$I_2 \neq \text{'d'}$  →  $I_2 = \text{'d'}$

$I_3 \neq \text{'!'}$  →  $I_3 = \text{'!'}$

SMT solver

→ SAT

bood

gaod

godd

goo!

Gen 1

Negate each constraint in path constraint  
Solve new constraint → new input