

---

# 20 ans de Recherches sur le "Software Model Checking"

1989

1994

2006

2009

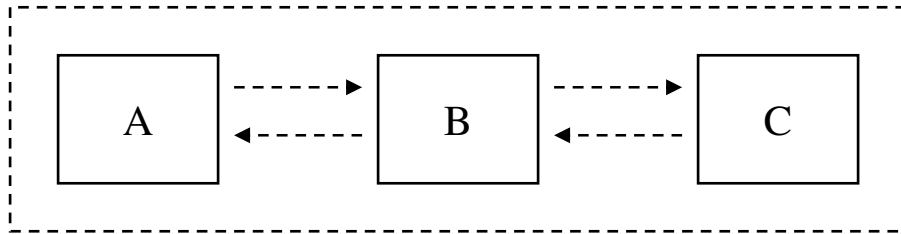
Université de Liège

Bell Labs

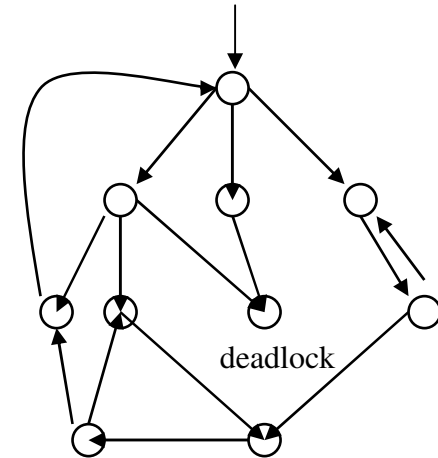
Microsoft Research

Patrice Godefroid

# "Model Checking"



Each component is modeled by a FSM.



- **Model Checking (MC)** is
  - check whether a program satisfies a property by exploring its state space
  - systematic state-space exploration = exhaustive testing
  - "check whether the system satisfies a temporal-logic formula"
- Simple yet effective technique for **finding bugs** in high-level hardware and software designs
- Once thoroughly checked, models can be compiled and used as the core of the implementation

# Problem: State Explosion!

---

## Example:

Initially:  $v_1=v_2= \dots =v_n=0$

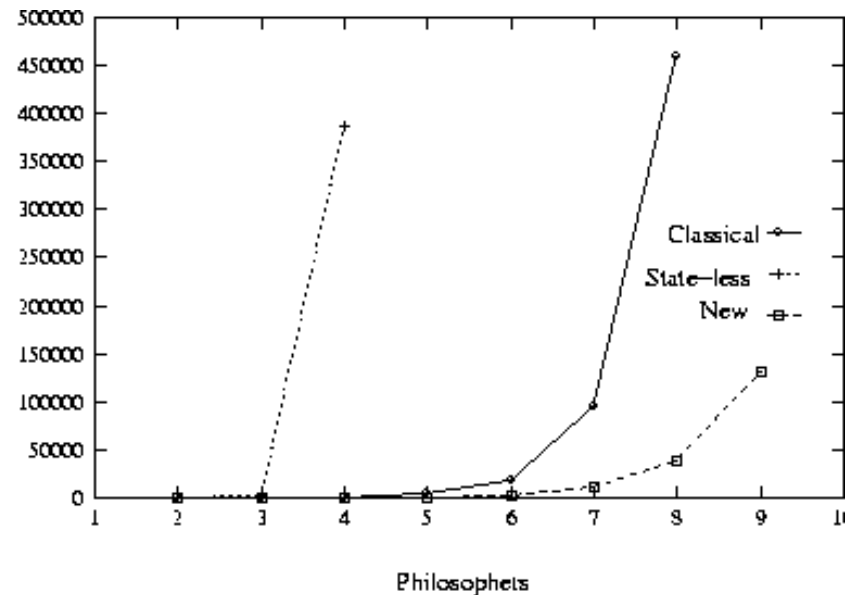
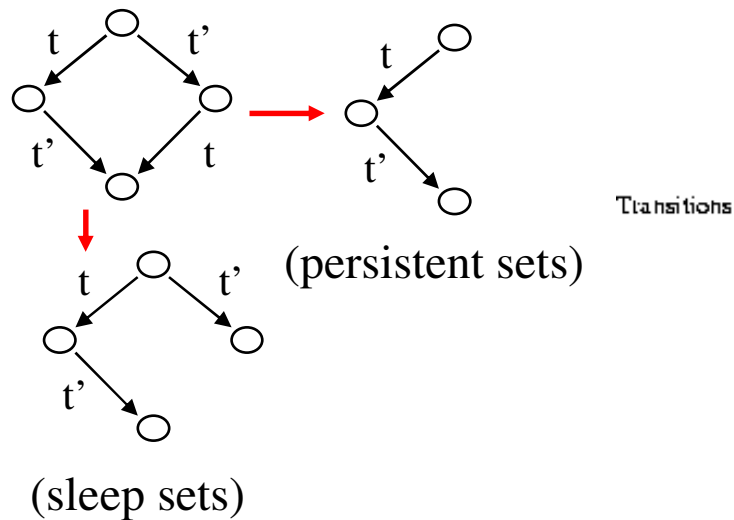
Process 1	Process 2	Process n
→ s1: $v_1:=1$ ; s1': stop	→ s2: $v_2:=1$ ; s2': stop	→ sn: $v_n:=1$ ; sn': stop

→ n! interleavings  
 $2^n$  states

→ **State Explosion**

# A Solution: Partial-Order Reduction

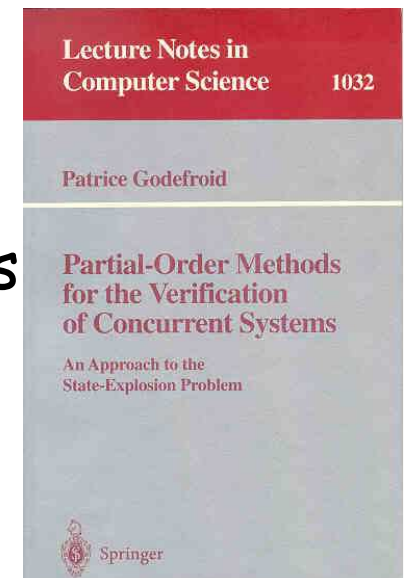
- Verification algorithms that avoid state explosion due to the modeling of concurrency by interleaving
- Examples:
  - 2 concurrent reads are commutative      reduction
  - But 2 concurrent writes are not      no reduction



# Impact

---

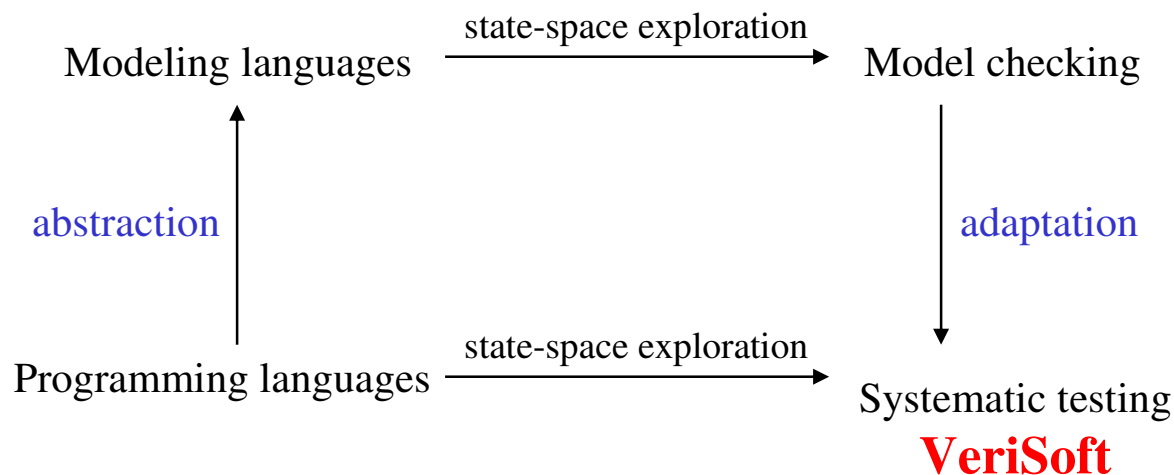
- We pioneered the development of partial-order reduction at the University of Liege (1989-1994)
  - We = Prof. Pierre Wolper, Didier Pirottin and me
  - With collaborator Gerard Holzmann (Bell Labs)
  - Other prominent contributors: Doron Peled (Technion, Israel) and Antti Valmari (Tampere, Finland)
- Developed first full-fledged tool with POR = "ULg Partial-Order Package for SPIN"
- Today, nearly all explicit-state model checkers implement POR in one form or another
  - Tens of tools
  - Hundreds of citations for our papers on the topic



# Problem: Model Checking of Software

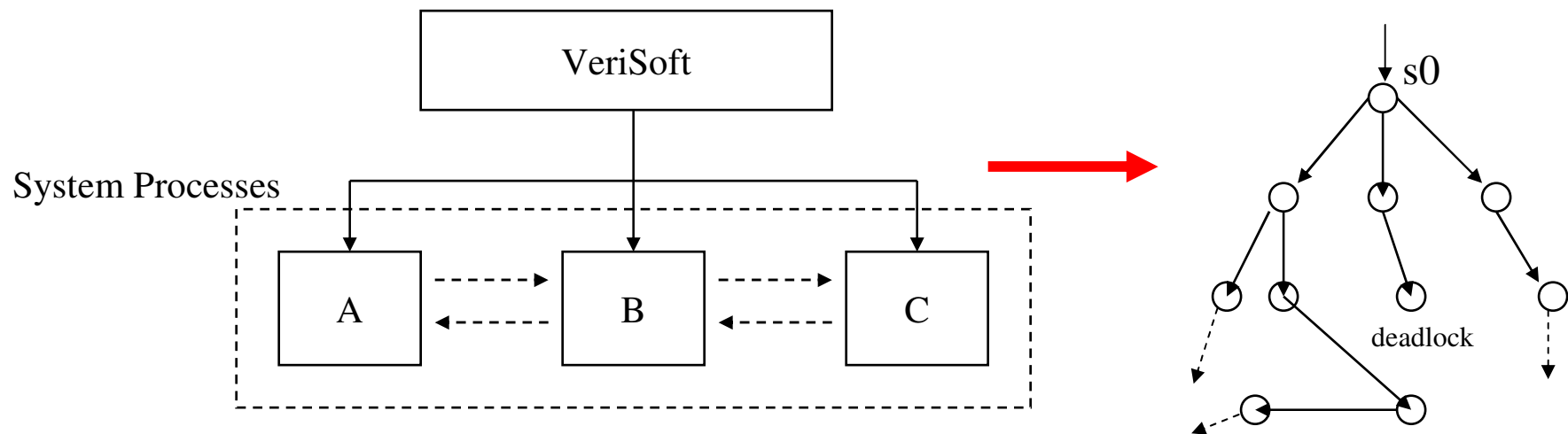
---

- 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:



# A Solution: VeriSoft = Systematic Testing

- State Space = "product of (OS) processes" (Dynamic Semantics)
- Systematically drive the system along all its state space paths (= automatically generate, execute and evaluate many scenarios)
- Control and observe the execution of concurrent processes by intercepting system calls (communication, assertion violations, etc.)
- From a given initial state, one can always guarantee a **complete coverage** of the state space **up to some depth**



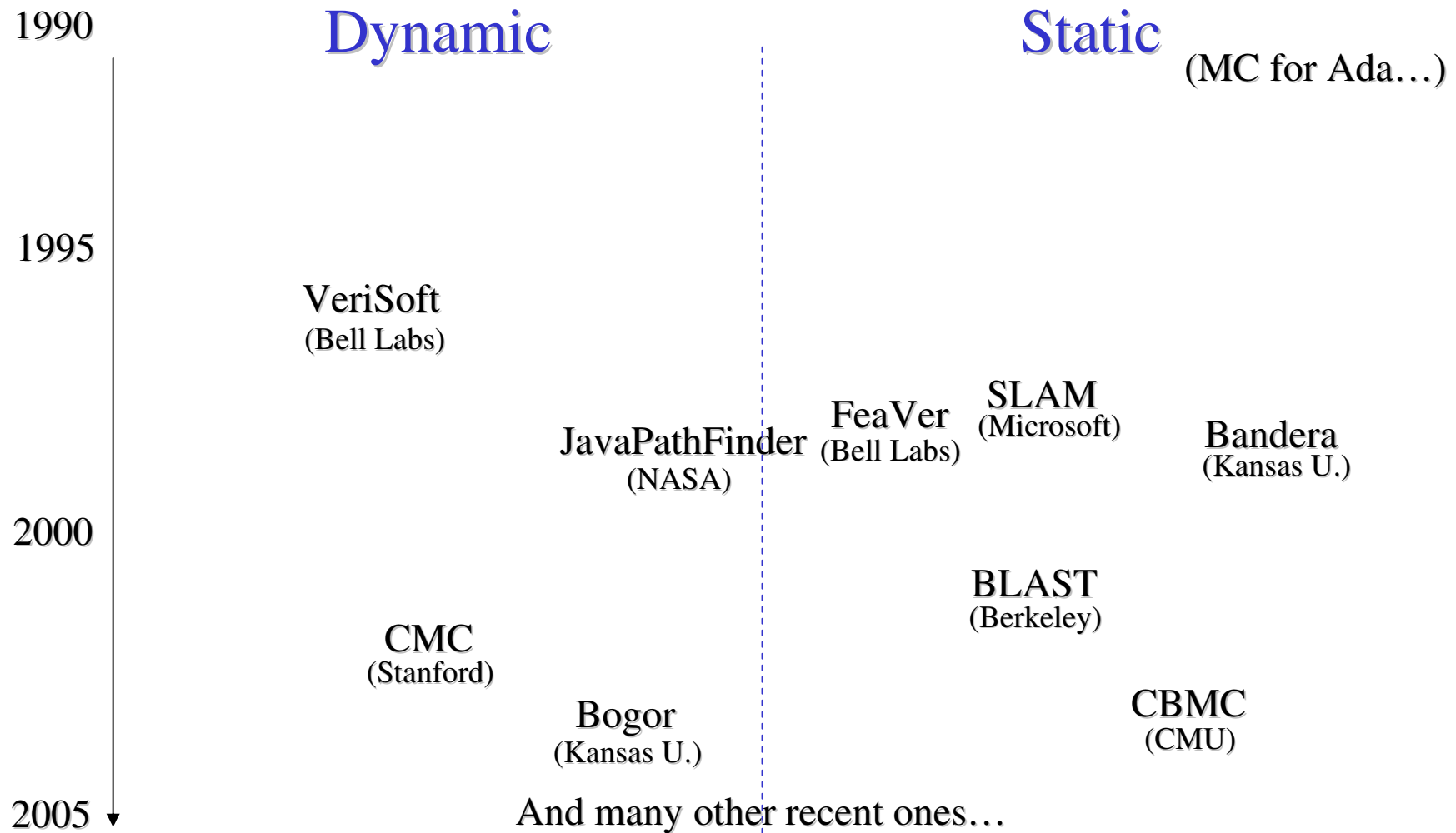
# Impact

---

- VeriSoft is the **first** systematic state-space exploration tool for concurrent systems composed of processes executing arbitrary code (e.g., C, C++, ...)
  - Many technical innovations: no static analysis (programming language independent), "VS\_toss(int)" to simulate nondeterminism at run-time, "state-less" search (no state encodings saved in memory), uses POR
- Examples of successful applications (at Lucent):
  - 4ESS Heart-Beat Monitor debugging and unit testing (1998)
  - WaveStar 40G R4 integration and system testing (1999-2000)
  - 3G Wireless CDMA call processing library testing (2000-2001)
  - Critical bugs found in each case ("-\$1M+ saved")
- VeriSoft is available outside Lucent since 1999
  - 100's of non-commercial (free) licenses in 25+ countries



# Software Model Checking Tools



# Problem: What about Data-driven apps?

---

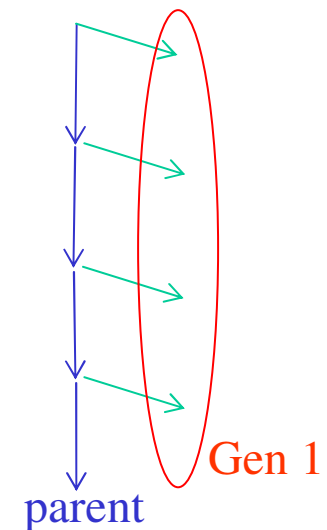
- How to systematically explore efficiently the state spaces of sequential programs to find bugs due to malformed inputs?
- Application: security testing at Microsoft
- Software security bugs can be very expensive:
  - Cost of each Microsoft Security Bulletin: \$Millions
  - Cost due to worms (Slammer,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



# A Solution: Whitebox Fuzzing

---

- Idea: mix fuzz testing with **dynamic test generation**
  - **Symbolic execution** to collect constraints on inputs
  - Negate those, **solve new constraints** to get new tests
  - Repeat "systematic dynamic test generation" (= **DART**)
- 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

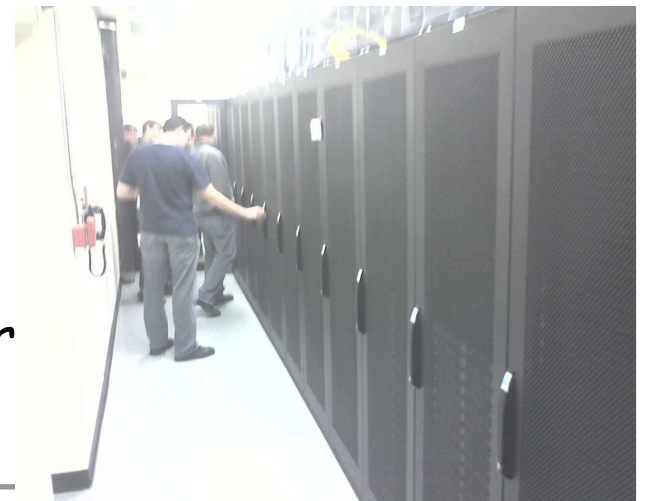


# Impact

---

Since April'07 1<sup>st</sup> release: many new security bugs found (missed by blackbox fuzzers, static analysis)

- Apps: image processors, media players, file decoders,...
- 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)
- Most bugs found by WEX Security team for Win7
  - Dedicated fuzzing lab with 100s machines
  - ~1/3 of **all** fuzzing bugs found by SAGE !
- SAGE = **gold** medal at Fuzzing Olympics organized by SWI at BlueHat'08 (Oct'08)
- Credit is due to entire SAGE team!
- Several other groups have now adopted our approach (10+ tools, 100s citations)



# Conclusion: Remerciements

---

- Université de Liège
- Professeur Pierre Wolper
- Tous mes collaborateurs ces 20 dernières années !
- L'AILg pour cet honneur